

The University of Toledo
COMMUNITY AND TECHNICAL COLLEGE
2801 West Bancroft Street
Toledo, Ohio 43606

The University of Toledo
COMMUNITY AND TECHNICAL COLLEGE
2801 West Bancroft Street
Toledo, Ohio 43606

System X^{TEN}

Laboratory Manual

— 6800 —

INDEX

TITLE	PAGE
Chapter 1	Introduction to Microprocessors 1
Chapter 2	Binary and Hexadecimal Numbers 10
Chapter 3	Keyboard Command and a Sample Program 18
	IV-A. Keyboard Functions 18
	H. Hello Can I Help You? 20
Chapter 4	Introduction to the System X 25
Chapter 5	Introduction to System X Programming 28
	III. Laboratory Exercise 30
Chapter 6	Introduction to Program Debugging 38
Chapter 7	Introduction to Input/Output 45
	III. Laboratory Exercise 52
Chapter 8	Understanding Data Arrays and Other Multiple Functions 64
	III. Laboratory Exercise 71
Chapter 9	Advanced Input/Output Formatting 84
	III. Laboratory Exercise 85
Chapter 10	Programming Techniques 100
	III. Laboratory Exercise 104
Chapter 11	Monitor Programs and the ASCII T68 Bug 123
	III. Laboratory Exercise 125
Chapter 12	The Peripheral Interface Adapter and the Asynchronous Communicator Adapter 135
	III. Laboratory Exercise 140
Chapter 13	Interrupts 152
	III. Laboratory Exercise 154

Chapter 1: Introduction to Microprocessors

Objective: To give the student a basic understanding of the history of computers and microprocessors, as well as the uses of microprocessors and the micro-minicomputer subsystems.

Student study requirements: Chapter 5 of BASIC MICROPROCESSORS and THE 6800 by Bishop.

Discussion: If we had to name the largest single item that has influenced the industrial world since the industrial revolution in the Eighteenth Century, it would have to be the microprocessor. In just a few short years since its introduction, the microprocessor has revolutionized the concept of digital electronics. Already, countless industries have capitalized on its low cost, versatility, and comparative simplicity.

It is the system's ability to perform functions by simply giving it a set of instructions to follow that has enabled designers to create more sophisticated and useful products with fewer and lower cost components than ever before.

Today, we can see the computer-on-a-chip being used in products that range from automotive applications and business machines, to home appliances and data communications. In 1976, manufacturers of microprocessors sold over \$100 million of these components, and sales are expected to pass the billion dollar mark by 1980. Or, to put it another way, by 1980 we will see a 1,000% increase in microprocessor use, compared to 1977.

How are all these microprocessors being put to work? Primarily as system controllers. This is accomplished by designing a small system into a larger one where it provides the control required for the larger system to function. Thus, the microprocessor becomes a discrete component within the framework of a total system. This is a considerably different application than the one associated with the general purpose minicomputer or larger main-frame computers used as stand-alone data processing systems.

A look at the history and development of the microprocessor shows that over the past 20 years, the speed at which calculations can be performed has increased by six orders of magnitude or, roughly, by a million times.

What's more, there has been a dramatic decrease in the cost of computation. Today, a \$1,000 minicomputer can nearly equal the capabilities of the larger units that cost as much as \$20 million just 15 years ago. By 1985, a medium scale minicomputer may cost less than \$100. Such economy is resulting in an amazingly increased ability to process information, and that, in turn, has led to an intellectual revolution, demanding knowledge of microprocessors at all levels of engineering.

The need to improve the speed and computing capacity of machines was recognized in the 1940s, but performance was limited by the kind of electronic technology then available. It relied on the vacuum tube and heat generated by the vacuum tube shortened the tube's operating life, and placed an upper bound on the size of computers. In fact, ENIAC, the first completely electronic computer, was developed at the University of Pennsylvania in 1945 and boasted 18,000 vacuum tubes. As you can see, larger machines were clearly impractical. It would have taken 24 hours a day just to find and replace defective vacuum tubes.

The computer was saved from a premature end by the invention of the transistor in 1947. The reductions in size and cost that have been achieved in the past 10 years were made possible by the development of the integrated circuit in the late 1950s.

Today, the designing of computers and other electronic devices is being transformed again by large scale integration (a process whereby tens of thousands of transistors and their interconnections are manufactured simultaneously). Through this technology, virtually all of the logic elements of a digital computer can be fitted into one chip of silicon that is no larger than a quarter of an inch per side. With the advent of the bubble memory, we can begin storing millions of bits of information on this same size package.

Let's compare the ENIAC and the System X:

	ENIAC	System X (average model)
Size	3,000 cubic feet	1/10 cubic foot
Power	140,000 watts	8 watts
Program Memory	16,000 bits	16,000 bits
Storage Memory	1,000 bits	6,000 bits
Clock Speed	100 Khz	1 Mhz
Tubes and Transistors	18,000 tubes	30 discrete devices
Resistors	70,000	28
Capacitors	10,000	25
Relays & Switches	7,500	1 keyboard
Mean time to failure	hours	years
Weight	60,000 pounds	4 pounds

Now that modern developments have given us a computer-on-a-chip, what does it all mean to the microprocessor industry? Present microprocessors vary in their detailed architecture depending on their manufacturer and, in some cases, on the particular semiconductor technology that is adopted. One of the major distinctions is whether all the elements of the microprocessor are embodied in one chip, or if they are divided among several identical modular chips that can be linked in parallel (the total number of chips depending on the length of the word the user wants to process: 4-bits **binary digits**, 8-bits, 16 bits, or more). Such a multi-chip arrangement is known as bit-sliced organization. A benefit of bit-sliced chips made by bipolar technology is that they are microprogrammable. They allow the user to create specific sets of instructions, which is a definite advantage for many applications.

The flood of microprocessors reaching the market, combined with the rapid rate of innovation, guarantees that any attempt to catalogue them will be instantly obsolete.

A more fruitful introduction to the micro market place is a study of the classification of systems according to capability and function. Along these two dimensions is a well defined upward progression in both hardware and software. In hardware the levels are chips, modules, breadboarding systems, small computer systems, full development systems, and multiprocessor systems. Software, the ability to communicate with and utilize hardware, is expanding beyond imagination and providing the industry with a powerful tool.

At the first level of the hierarchy are the microprocessor chips, representing the large scale integration of tens of thousands of individual electronic devices, transistors, diodes, resistors, and capacitors. You will also find more specialized chips at this level:

Random Access Memories (RAMs), Read Only Memories (ROMs), Programmable Read Only Memories (PROMs), Input/Output (I/O), interfaces, and others. The cutting edge of the technology works most directly at the chip level providing, for example, RAMs of ever higher storage capacity. Currently the most advanced commercially available RAM can store 64K.

The compatibility of chips and chip families made by different manufacturers varies widely. For example, the microprocessors of different builders are not physically interchangeable, but some memory chips are. For instance, the 6800 chip in the System X is compatible with the entire Motorola family, as well as with some of the chips in the Intel family.

The second level of the hierarchy — the module and breadboarding systems — represents the simplest form of computer system. It can be created by combining a microprocessor with a limited array of memory chips (RAMs and ROMs) and input/output chips. In order to communicate with such a minimal system, the user will also need a simple device such as a numeric keyboard, plus a device capable of displaying or recording the computer output. Such single board systems are useful for introductory teaching purposes, or they can serve as breadboarding prototypes for more sophisticated systems. A beginner can learn the fundamentals of microprocessor programming with only a modest investment.

The next level in the hierarchy of capability and function contains the small computer systems that are prepackaged as stand-alone units. Unlike the single board modules, they have a larger power supply, the capability for memory expansion, and room for a series of plug-in interface modules. Some of the more powerful single board modules can be expanded with the appropriate hardware to create a single-box computer system. All the small computer systems have software capabilities approaching the sophistication of those found in much larger conventional systems. They also provide an interface for a cathode-ray-tube (CRT) or a keyboard display console. Many of the small systems can also be interfaced to such peripheral devices as floppy disk memories, tape cassettes, paper tapes, and line printers. With such enhancements, a small computer system could serve as a full development system. The small computer system is growing so fast that by the time you finish this, a new development will have been completed — probably in the area of greatest growth which is industrial control and processing.

The debut of the fully automatic factory has been forecast for decades, but it has yet to arrive. The reason for the delay? Automation involves far more than the development of sequential machines. It calls for feedback mechanisms that first sense anomalies in the system, and then analyzes them, and takes appropriate corrective action. In a large manufacturing process, thousands of things can go wrong and send a nonadaptive system into a spin. For example, in the motion picture "Modern Times," Charlie Chaplin takes a work break to be fed lunch by an automaton that keeps putting the food everywhere but in his mouth. (Charlie was a little short for the machine, and it could not adapt.)

It will be a long time before the majority of production processes can be truly automated. Nevertheless, as individual machines are made "smarter," we can expect them to cope successfully with an ever widening range of problems. A good illustration is seen in the evolution of traffic signal systems. The first automatic traffic signals were purely sequential: they were fixed in their timing, and were independent of one another. The flow of traffic was greatly aided many years ago by both synchronizing the system, and changing its timing to accommodate the variations in traffic flow that occur during rush hours. Even so, anomalous local traffic flows often called for manual intervention.

Today, traffic signals can sense and count the number of cars in the mainstream of traffic, and even the accumulation of cars in the access lanes. They can be programmed to adjust their own timing to keep traffic flow at an optimum level. Adaptive control of this kind is all in a day's work for a microprocessor.

A similar evolution can be seen in machine tools. Such devices as turret head lathes and screw machines have routinely performed complex sequential operations for many years. In the past, programs were part of the hardware: cams and ratchets sequenced the machine through a series of steps to manufacture a particular part. The variety of sequences was necessarily limited, however, and the cost of new hardware programs ran high.

Then in the early 1960s, numerically controlled machine tools made their appearance (machines coupled to the minicomputer). Such machines could mill, drill, bore, and tap holes anywhere and at any angle on a work piece of almost any shape. The appropriate tool head was selected automatically by a punched paper tape program that was read by an electronically controlled computer controller. The same program positioned the tool in three coordinates, and

dictated the speed of the tool spindle and the depth and rate of the cut. To revise the final shape of the work piece, or to make an entirely different one, it was necessary only to amend the old punch tape in order to provide a new one.

Numerically controlled machines of this kind filled an important need in intermediate volume production where the output of work pieces was substantial, yet not high enough to warrant a mass production line (e.g., where each of a series of machines performs only one operation). However, computer coupled machines were quite expensive, and a substantial portion of that cost was in the computer controller.

Today, microprocessor controllers no larger than desk top calculators are taking the place of the larger first generation controllers, and cassette tapes are replacing the rolls of punched paper tape. Also, micro-electronic controllers add very little to the cost of the machine tools.

We can expect to see further progress in microprocessor controls in our own automobiles. The microprocessor is now making it economically feasible to extend electronic controls to such functions as ignition timing and carburetion. The motivation to develop such systems is strong, because they help with fuel economy and aid in meeting pollution standards.

Good ignition timing is a function of two things: the speed of the engine, and the vacuum in the intake manifold. The mechanisms used in today's cars can only approximate the optimums, but the application of microprocessors can ensure correct timing at all speeds and loads. Ideally, these devices should play the role of monitors, and not be responsible for delivering each charge of fuel, or for firing each spark. Such a monitor would compare actual performance with ideal performance, and make the necessary adjustments to the existing basic system.

If the microprocessor were used in this way, it would have the advantage of a greatly reduced data rate, ordering only a few corrections per second. Furthermore, it would be fail safe. If the microprocessor were to go out, the engine would still operate: its performance would simply be reduced to the level that prevails today. The same microprocessor would continuously monitor the health of the vehicle, and report in plain English on a LED display any troubles such as low fuel, overheated engine, or overheated or worn brakes. The cost of such a system lies not so much in the microprocessor components as in the sensors needed to feed the data to the microprocessor, and in the actuators needed to carry out its instructions.

It appears that control systems and their associated sensing components are becoming more and more like the human nervous system. Microprocessors in satellite controllers, by enabling "intelligence" to be distributed broadly throughout the control system, greatly reduce the amount of data that must be transmitted over data links and handled by the central computer. In the human nervous system, much control is also at the local level, and much processing of input data is done in ganglia. The nervous system is a network of sensors and microprocessors connected by data links to a central computer. Both the satellite processors and the central computer have a lot of firmware (subroutines that enable us to do 99% of what we do without "thinking," or without invoking our highest control centers). Here, too, micro-electronic control systems resemble the human control system. The central computer merely has to order an operation to be done: it need not tell each smaller machine every step of how to do it.

The examples cited are only a few of literally hundreds of ways in which the microprocessor will improve and simplify the operations of the machines that serve us daily. Integrated circuit technology has so reduced the cost of computation and of logical systems that most of the devices we use can now be much more cooperative and intelligent than they have been in the past. These decades will be recognized in the future as the beginning of the "Intelligence Revolution."

Chapter 2: Binary and Hexadecimal Numbers

Objective: To introduce the student to binary, octal, and hexadecimal numbering systems.

Please note: Through outside study, the student should familiarize himself with binary and hexadecimal numbering systems. These numbering systems are the very heart of the microprocessor, and a programmer **must** know how to use them.

Student study requirements: Chapter 3 of BASIC MICROPROCESSORS and THE 6800 by Bishop.

Discussion: There are certain common characteristics in number systems: each number system has a base (radix) which refers to the number of separate numerals or digits used in the system. The decimal number system is called base 10 because of the 10 separate digits, 0 through 9, that are used. Binary means base 2 because the only digits that are used are 0 and 1.

Theoretically, a number system can be developed on any whole number or integer, but we will concentrate on the binary, octal (base 8), and hexadecimal (base 16) systems.

011_2

127_8

$1A4_{16}$

36_{10}

Numerals: The value of a numeral within a number depends upon two things: the numeral itself, and its place within the number. Through long practice and common agreement, the decimal numeral 9 is considered to have a larger value than the numeral 5. Whenever we count, we simply use each numeral in increasing value until we reach the highest numeral in that number system. Then we repeat the numerals in that position with the next higher numeral in the position to the left. The technique with decimal numbers seems almost too commonplace to bear repeating, but the point is that the same principle is used no matter what number system is employed.

Decimal	Binary	Octal	Hexadecimal
0	000000	0	0
1	000001	1	1
2	000010	2	2
3	000011	3	3
4	000100	4	4
5	000101	5	5
6	000110	6	6
7	000111	7	7
8	001000	10	8
9	001001	11	9
10	001010	12	A
11	001011	13	B
12	001100	14	C
13	001101	15	D
14	001110	16	E
15	001111	17	F
16	010000	20	10
17	010001	21	11

and so it continues.

One of our reasons for showing the octal system, as well as the binary and hexadecimal, is that among major computer manufacturers, the Honeywell computers normally use octal representation for the binary numbers in internal storage. You will also find that the use of the octal system makes conversion from the decimal system to hexadecimal a simple step.

Conversion: You may search a number of textbooks and find many different methods of conversion and the theory behind each, and they all work. However, since I believe in doing things as easily as possible, I will show you the method I have found to be the easiest. Once you convert to an octal base, you have a direct conversion to binary and hexadecimal, so let's convert a number (180 base 10) to octal:

Example: 180 ---
 10 8

Setup as follows:

8	180	4
8	22	6
		2

Take the base 10 number 180, place it in a box, divide that number (180) by the number 8, place the answer under (180) which is 22, with the remainder on the right side. Now divide 22 by the number 8, again placing the answer underneath, and putting the remainder to the right. The answer is 2, with a remainder of 6, read from the bottom up giving 264. So, we now see that 180 in the base 10 equals 264 in the base 8. I fully realize that there will be some confusion to begin with, but after a couple of examples you will catch on quite quickly.

Examples:

1. 499 base 10 to base 8 = 763₈

8	499	3
8	62	6
		7

2. 1245 base 10 to base 8 = 2335₈

8	1245	5
8	155	3
8	19	3
		2

$$\begin{array}{r}
 264 \\
 \times 8 \\
 \hline
 16 \\
 + 6 \\
 \hline
 22 \\
 \times 8 \\
 \hline
 176 \\
 + 4 \\
 \hline
 180
 \end{array}$$

Just to see if the system works, let's try to convert from base 8 to base 10. Follow the example. Take the Most Significant Bit and multiply by 8, the result being 16. Take the next MSB and add to 16, giving 22. Now multiply by 8 again. This will give you 176 at the Least Significant Bit (which is 4), giving you 180 for the answer. Once you have added the LSB, the conversion is complete. Convert the answers you received from the other problems to see if you answered them correctly.

Now that we have found how to convert from decimal to octal, it is easy to convert first from octal to binary, and then to hexadecimal. Let's take the number we have worked with (180_{10} or 264_8) and convert it to binary.

2
010

6
110

4
100

The binary value under the octal value is a direct conversion as found on page 32 of BASIC MICROPROCESSORS and THE 6800 by Bishop. So, 180 base 10 equals 264 base 8 equals 010110100 base 2, or properly written will read:

$$180_{10} = 264_8 = 010110100_2$$

You will notice that on the conversion from octal to binary, the binary word is written in a 3-bit format, and the conversion chart shows that hexadecimal is written in a 4-bit format. So, to convert from binary to hexadecimal format, all we have to do is change from a 3-bit to a 4-bit format.

Example: 010 110 100 equals 0 1011 0100, the change being made from right to left. Now take the number in the 4-bit format, and convert it directly, using the charts on page 32 of BASIC MICROPROCESSORS and THE 6800 by Bishop.

Example:

0	1011	0100
0	B	4

Thus, the number 180 in base 10 equals 264 in base 8 equals 010 110 100 in base 2 and equals B4 in base 16.

Now for another example. Let's say that you have to use the number 4011 in the System X. The unit uses only the hexadecimal system, so we must convert.

8	4011	3
8	501	5
8	62	6
	7	

7	6	5	3
111	110	101	011
1111	1010	1011	
F	A	B	

To place the decimal value 4011 in the System X, we would use the base 16 hexadecimal value of FAB.

Since the System X is a byte oriented machine, this is an excellent time to define the byte. One hexadecimal digit represents a 4 digit (bit) binary word. Two hexadecimal digits (8-bits) represent a byte, so 01101011 base 2 equals 6B in base 16 is a byte word or 8-bits, and a 2-byte word would be 2F4A, or 0010 1111 0100 1010 in binary.

Next convert 2F4A (2-byte word) in the System X to its equivalent decimal value, using the chart on page 32 of BASIC MICROPROCESSORS and THE 6800 by Bishop. First convert to binary:

2	F	4	A
0010	1111	0100	1010

Then to the octal binary equivalent:

0, 010, 111, 101, 001, 010 = 010 111 101 001 010

Then convert to octal:

27512

Now from octal to decimal: Note: A 4-function calculator may be used for this.

A handwritten calculation showing the conversion of octal 27512 to decimal 12106. The calculation is performed in columns, with each column representing a power of 8. The digits of the octal number are 2, 7, 5, 1, and 2 from left to right. The calculation proceeds as follows: 2 is multiplied by 8 to get 16; 7 is added to 16 to get 23; 23 is multiplied by 8 to get 184; 5 is added to 184 to get 189; 189 is multiplied by 8 to get 1512; 1 is added to 1512 to get 1513; 1513 is multiplied by 8 to get 12104; and finally, 2 is added to 12104 to get the final result, 12106. Arrows indicate the flow of the calculation from one column to the next.

27512
x8
16
+7
23
x8
184
+5
189
x8
1512
+1
1513
x8
12104
+2
12106

If this system is difficult for you, switch to a system that feels comfortable, and practice. It is vitally important to be able to convert from one number base to another as rapidly and accurately as possible.

You are probably asking yourself, why worry about number systems at all? Why not use the old fingers and toes as we always have? The answer to that problem is very simple. A logic system (microprocessors, computers, switches, etc.) can work only with 1's and 0's (either on or off), so the use for the binary system.

The hexadecimal system serves as an easy method of stating a binary word. Just trying to say or use the word 0010111101001010 will convince you how much easier it would be to use 2F4A, and it saves space too. The reason you will need an understanding of both binary and hexadecimal systems will be made

clear once you start programming. You will find that much of a program requires you to look at only one or certain bits within a byte (word), so you must understand how to use binary. A good example of the use of binary is in subtraction. Most digital systems cannot subtract: they can only add. So, you are going to need a method of adding that gives the results of subtraction. Sound confusing? Actually, it is rather simple. A complement is used for subtraction, as well as other logical functions. A complement is something that is used to complete something else. The complement of an angle is the amount to be added to that angle to make a straight line.

In number systems, we find two complements. The first is the amount necessary to complete a number made up of the highest value digits in the number system. In the decimal system, this would be the difference between any given number and all 9's.

For example, the 9's complement of the number 374 is 999 minus 374, or 625.

The second type of complement is the difference between a number and the next higher power of the number base. For example, the next higher power of 10 above 374 is 1,000. The difference between 1,000 and 374 is 626. This is referred to as the 10's complement in the decimal number system. Complements are used in subtraction. If we disregard the "carry" beyond the size of the original number, and add the 10's complement of any number, it will have the same effect as subtracting the original number.

Example: Subtraction

591	591	
<u>-374</u>	<u>+626</u>	10's complement of 374
217	×217	

We use the complement for subtraction, and the System X works in binary, so let us look at binary complements. They exist in two forms referred to as 1's and 2's complements. The 1's complement is the difference between any binary and all 1's while the 2's complement is the difference between the binary number and the next higher power of 2.

Now let's look at a quick and easy way to form the 1's complement. Simply change each 0 in the original number to a 1, and each 1 in the original number to a 0. For clarity's sake, it is best to have several 0's leading to the original number. The leading 0's will always become 1's in the complement. Usually it is easier to form the 1's complement and add 1 to the result than it is to subtract the original number from the next higher power of 2, because you will have to borrow.

Simply stated, to reach 2's complement of any binary number, change all 1's to 0's and all 0's to 1's, and then add 1.

Example:	Take	0010	1011
	1's complement	1101	0100
	add		+1
	2's complement	1101	0101

Now let us do subtraction using 2's complements:

7	111		100	111
<u>-4</u>	-100	2's	011	<u>100</u>
3			<u>1</u>	011 = 3
			100	(disregard carry)

We recognize at this point that we have only scratched the surface of numbering systems. However, students should complete additional work, and also study those texts previously mentioned.

Chapter 3: Keyboard Commands and A Sample Program

Objective: To familiarize the student with the System X keyboard, and to explain the loading of a sample program.

Student study requirements: The USER'S MANUAL

Discussion

- I. The System X is operated with 16 keys (0-F) on the hexadecimal keyboard (See drawing of keyboard in the USER'S MANUAL.)
- II. The student should plug the System X into a 110V - AC plug. The off/on switch is located on the rear right side - depress and turn system on. Also depress the RESET key and depress and release any key for control. The display will show ASCI UP.
- III. RESET is initialized by depressing the RESET key, and depressing and releasing any key for control. Try it. What happens? The display will show ASCI UP. The RESET must be used whenever you change modes.
- IV. Let us now examine some of the keyboard functions.
 - A. "A" key doubles as AUTO command (See the USER'S MANUAL) Depressing this key causes the contents to be cleared, and the address to be automatically incremented.
 1. Depress the "A" key. The display will show --- b b (b = Blank).
 2. Now address the System X by depressing 0, 1, 0, and 0.
 3. The display will show 0100 --.
 4. Now depress the "F" key twice. The display will show 0101 --. We have just loaded FF into memory location 0100 of the System X.
 5. Depress the "E" key twice. What happened? Notice on the next entry what the right LEDs do when the second key is released. The display now reads 0102 --.
 6. Depress the "d" key once.

7. Now depress the "D" key and hold it down. the display reads 0102 dd, which tells you that dd will be stored in memory location 0102 once you release the key.

8. Release the key. The display will read 0103 --.

9. Depress any two keys you desire and load the memory to location 0110. Be sure to keep a copy of what is located in these locations for you will look at this data in future steps.

B. Depress RESET and any key for control. The System X should always be RESET when going from one function to another.

1. Now depress the "E" key, which doubles as EXAMINE. This command allows you to enter any address and examine its contents. (See USER'S MANUAL.) The display shows ---- b b.

2. Depress 0100. The display shows 0100 FF, which is the data we entered previously.

C. Now depress the "F" key, which doubles as FORWARD. This command allows you to increment the displayed address forward through memory and examine the contents. (See the USER'S MANUAL.)

The display shows 0101 EE.

1. Depress "F" again. The display will show 0102 dd.

D. Now depress the "B" key which doubles as BACK. This command lets you decrement the displayed address back through memory and examine the contents. (See the USER'S MANUAL.) The display will show 0101 EE.

1. Depress "B" key again. The display will show 0100 FF.

2. Now that you have been introduced to the "E" "F" and "B" keys, try going up through memory by using the "F" key and checking the data you have previously entered.

3. Now go back through memory checking the same location. When you reach 0110, step back to location 0102. The display will read 0102 dd.

E. Depress the "C" key which doubles as the CHANGE command. This command allows you to modify the contents of the one memory location. (See the USER'S MANUAL.) The display shows 0102.

- 1.** Depress the "A" key twice (referred to as enter AA into 0102). The display will show 0102 AA.

- 2.** Now depress the "B" key. (You may go in either direction.) The display will show 0101 EE.

- 3.** Depress "C" and enter bb. The display will show 0101 bb.

- 4.** Now, by depressing the "F" key, step forward to location 010A and change it to read CE. Were you capable of entering the data?

- 5.** Try on your own to load data "A", examine at forward and backward "F" and "B", and change the data "C".

F. The "P/C" (7) key doubles as the set Program Counter key. This command lets you enter the starting address of the program, and then that program will be executed by depressing the "do" (d) key.

G. You will have noticed by this time that all the keys of the ASCII System X have dual functions. You will cover these additional key functions as you proceed through the laboratory experiments. Additional information may also be found in the USER'S MANUAL.

H. Now that we have learned a few of the functions of the System X, let us load the program "HELLO CAN I HELP YOU?" This starts on page of this manual. I realize that this is a long program for you at this point, but it will give you a program that does something and familiarizes you with the keyboard.

- 1.** Load the program. Now take your program listing and notice that memory locations 0060 through 0088 contain data, while locations 0089 through 00bA contain the program. This program is written in HEX, but you will not have to do any conversion.

- 2.** Depress RESET on the System X and any key for control. The display shows ASCII UP.

- 3.** Depress "A" key. Display ---- b b.
- 4.** Depress 0060 Location of first data. Display 0060 --.
- 5.** Now enter first data. Depress "0" then "1" keys. Display reads 0061 --.
- 6.** Enter data for address 0061. Display reads 0062 --.
- 7.** Continue to enter the hexadecimal code for each memory location 0062 through 00bA.
- 8.** Check your entries after you have completed all locations by reset "E" "F" and "B" keys. If you have any incorrect data, use the "C" key and correct the data.
- 9.** Once you are satisfied that all the data and the program are in the correct memory location, you are now ready to run the program.
- 10.** Depress RESET, depress and release any key.
- 11.** Depress "SET PC" key. Display will show ---- PC.
- 12.** Enter the starting address of the program 0089. The program will start when you depress and release the "DO" key, and the display will read "HELLO CAN I HELP YOU??" in marquee fashion.
- 13.** If your program fails to work properly, press RESET and then go back and check your program. Remember, you must always RESET when changing modes.

CODING

Problem Display Marquee Method

FORM

By _____ Date 6/27/79

Location	Hex	Instr.	Operand # , X	Comment
0060	0	01		—
0061	0	01		—
0062	0	01		—
0063	0	01		—
0064	0	01		—
0065	0	01		—
0066	6	E		H
0067	9	E		E
0068	1	C		L
0069	1	C		L
006A	F	C		0
006B	0	0		Space
006C	0	0		Space
006D	0	0		Space
006E	0	0		Space
006F	9	C		C
0070	E	E		A
0071	E	C		N
0072	0	0		Space
0073	0	0		Space
0074	0	0		Space
0075	6	0		I
0076	0	0		Space
0077	0	0		Space
0078	0	0		Space
0079	6	E		H
007A	9	E		E
007B	1	C		L
007C	C	E		P
007D	0	0		Space
007E	0	0		Space
007F	0	0		Space

CODING

Problem Display Marquee Method

FORM

By _____ Date 6/27/79

Location	Hex	Instr.	Operand # , X	Comment
00A0	0 0 8 0	7 6		Y
	0 0 8 1	F C		0
	0 0 8 2	7 C		U
	0 0 8 3	0 0		Space
	0 0 8 4	C A		?
	0 0 8 5	0 0		Space
	0 0 8 6	0 0		Space
	0 0 8 7	0 0		Space
	0 0 8 8	0 0		Space
	0 0 8 9	7 F	C L R 0 0 9 4	Clear FETCH START
	0 0 8 A	0 0		
	0 0 8 B	9 4		
	0 0 8 C	C E	L D X	Load index register
	0 0 8 D	0 0		
80	0 0 8 E	6 0		
	0 0 8 F	8 6	L D A A # \$ 4 A	Load Display Digits
0020	0 0 9 0	4 A		
	0 0 9 1	9 7	S T A A	Store to Counter
36	0 0 9 2	9 6		
	0 0 9 3	A 6	L D A X 0 0 , X	Load Data to be displayed
0034	0 0 9 4	0 0		Fetch Pointer will change
	0 0 9 5	9 7	S T A A	Data Counter
0036	0 0 9 6	0 0		Counter Pointer will change
	0 0 9 7	7 C	I N C	Increment Fetch Pointer
	0 0 9 8	0 0		
124	0 0 9 9	9 4		
	0 0 9 A	7 C	I N C	Increment Data Pointer
	0 0 9 B	0 0		
26	0 0 9 C	9 6		
	0 0 9 D	d 6	L D A B	See if all digits loaded
36	0 0 9 E	9 6		
	0 0 9 F	C 1	C M P B \$ 5 0	

CODING

Problem Display Marquee Method

FORM

By _____ Date _____

Location	Hex	Instr.	Operand # , X	Comment
00A0	50			
00A1	26	BNE		No, Go back to load data
00A2	F0			
00A3	86	LDA	A	Load Timer
00A4	F1			
00A5	97	STA	A	Store Timer Temporary
00A6	96			
00A7	b d	JSR		Go to Display
00A8	F8			
00A9	0C			
00AA	7C	INC		Increment Timer
00AB	00			
00AC	96			
00AD	26	BNE		Timer is zero, no go to display
00AE	F8			
00AF	96	LDA	A	Yes, Load Fetch to A
00B0	94			
00B1	80	SUB	A	Sub 5.
00B2	05			
00B3	97	STA	A	Store back to Fetch
00B4	94			
00B5	81	CMP	A \$ 24	Is Fetch Equal to 24
00B6	24			
00B7	26	BNE		No, Go to, Load Index
00B8	d3			
00B9	20	BRA		Yes go to Start
00BA	CE			
00BB				To run program, refer to page 21,
00BC				line #10.
00BD				
00BE				
00BF				

Set PC to 0089, Depress d0

Chapter 4: Introduction to the System X

Objective: To familiarize the student with the physical layout of the system; to coordinate the schematic to the actual components (chips); and to explain the relationship of the memory map to various programming techniques.

Student study requirements: Chapter 9 of BASIC MICROPROCESSORS and THE 6800 by Bishop, and the LABORATORY MANUAL.

Discussion

- I. Introduction to the ASCI System X circuit board layout.
 - A.** The student should turn to the component layout diagram in the back of the USER'S MANUAL page 21. The student should have a unit in full view with the cover lifted to observe the physical properties of the System X.
 - B.** The student should locate the microprocessor (M6800P) unit (MPU). It is the heart of the unit and located to the right center of the board (the largest chip — a large scale integration, 40 pin dip). The MPU uses six (6) internal registers, including the A and B accumulators. The actual chips and pin connections will be discussed at a later time.
 - C.** To the right of the MPU are the data bus drivers (8T26). These integrated circuits are the handshaking devices between the M6800 and the outside world, or in other words, amplifiers to connect the MPU to the data bus.
 - D.** Directly above the bus driver is the clock circuit consisting of a 6875 chip and a 4Mhz crystal. This clock times the system at 1Mhz to synchronize all signals. The 555 chip is the RESET control.

E. To the left of the MPU you will find four (4) 74LS368 chips. These are the address buffers, or like the data buffers, are amplifiers to connect the MPU to the address bus.

F. The remaining chips along the bottom edge are the address decoding and control units. Notice two (2) sockets are blank and held in reserve for the addition of 32K control.

G. In the center of the board are located ^(2 IF USING MICROTAPE) ~~one (1)~~ 2716 EPROMS. (The two small switches allow the 2716 to be changed to 2708s). These Erasable Programmable Read Only Memories contain the AU keyboard routines or the ASCII T68 bug monitor.

H. To the left of the EPROMS you will find two (2) 2114 RAM (Random Access Memory) chips. These are the memory units and they are 1K 4-byte units, or on board is 1K of 8-bit memory.

I. Above the ROM/RAM locations are the three (3) 6821 Peripheral Interfact Adapters (PIA) that provide the interface capabilities to all peripheral equipment. The center unit is for control of the LEDs on the control unit. The leftmost unit is connected to an unassigned edge connector for future expansion. The last unit (to the right) is the PIA for the on board keyboard. Note that the lamp drivers are **not** on the mother board, but are part of the subassembly with the keyboard.

J. The left side of the unit contains the chips for ACIA baud rate and serial I/O. The fixed baud rate of 110 is used although a variable is possible.

K. The unit contains a power supply with an off/on switch, a fuse, and a power cord. The entire system is mounted on a sturdy frame with a top cover that is hinged for convenience. The keyboard and the display are covered in another chapter.

II. Memory Map

A. The student should turn to the memory map page 17 in the USER'S MANUAL. Although the memory map information may seem out of place at this time, the student must understand about memory locations before programming is possible. The primary configuration of this unit contains 1K of memory available to the user except for \$60 locations used for scratchpad.

B. User RAM locations start at 0060 and extend to 03FF, with expansion to E000.

C. E000 to ~~E400~~^{3FF} is the keyboard PIA space.
E400 to ~~E800~~^{7FF} is the display PIA space.
E800 to ~~EC00~~^B is spare PIA.
EC00 to ~~EF00~~^{FFF} is for cassette interface/ACIA
F000 to F7FF is PROM

D. The user must remember not to use memory locations other than the locations assigned. Other locations will create problems.

Chapter 5: Introduction to System X Programming

Objective: To introduce the student to the M6800 instruction set and addressing modes; M6800 addressing methods; entering and executing programs; direct addressing; and LDA, COM, STA, NEGA, and SWI instructions.

Student study requirements: Pages 79-85 (paragraphs 7.2. - 7.8), 103, 111, 115, 124, and 128 of BASIC MICROPROCESSORS and THE 6800 by Bishop; pages 32, 45, 63, and 67 of the M6800 PROGRAMMING MANUAL.

Discussion

- I. As an introduction to the programming, and as we begin exercises in the LABORATORY MANUAL, it is well to mention that all programs in the LABORATORY MANUAL are written both in assembler language and in machine language. Assembler language is simply a programming improvement to assign a name to each instruction code. This instruction code name is called mnemonic or memory aid.

Example: ADD (mnemonic) 9B (instruction set code)

Assembler is a task we can assign to the microprocessor. It translates a user program written with mnemonics into a machine language program (object program) which the microcomputer can execute. The assembler's input is in the mnemonics source program, and its output is in the hexadecimal object program.

The best explanation for the introduction of assembler in the LABORATORY MANUAL is for the student to recognize its importance as a programming aid, and the fact that most microprocessor programs in industry are first written in assembler language on a large microcomputer, and then translated into machine language for use in the microprocessor. The assembler makes programs easier and faster to write, and upon completion of the LABORATORY MANUAL, a student should have a good background in assembler language programming, as well as becoming proficient with machine language.

II. Addressing Modes

A. In addition to having 72 instructional codes, the System X also has six addressing modes. By looking on pages 78 - 85 (paragraphs 7.2 - 7.8) of BASIC MICROPROCESSORS and THE 6800 by Bishop, you will notice that six addressing modes are listed.

1. Six Modes

- a. Inherent/Accumulator (Implied)
- b. Immediate
- c. Direct
- d. Extended
- e. Indexed
- f. Relative

2. During the course of the laboratory exercises, each addressing mode will be covered in-depth, along with an example.

B. In the Chapter 5 laboratory exercise, we will use both the implied (inherent/accumulator) mode and the direct mode.

1. Implied Mode: This addressing mode has only 1-byte instructions, and therefore, does not require addressing a memory location or the addressing information is contained in the instruction. An example would be COMA, which means complement the A accumulator (explained in detail later in this chapter).

Example: MEMORY ADDRESS MEMORY CONTENTS

\$00 \$96 (LDAA)

\$01 \$40 (address)

Load the contents of memory address \$40 into the accumulator A.

- B. COMA (Complement Accumulator A):** This complements the contents of the specified accumulator. No other status bit or register's contents are affected. (See page A32 of the M6800 PROGRAMMING MANUAL)

Example: MEMORY ADDRESS MEMORY CONTENTS

\$02 \$43 (COMA)

Suppose the accumulator A contains \$2B or 0010 1011₂.
Upon this instruction, it would change to 1101 0100 complement.

- C. STAA (Store Accumulator A):** This instruction stores the contents of the selected accumulator into the specified memory location. (See page 63 of the M6800 PROGRAMMING MANUAL)

Example: MEMORY ADDRESS MEMORY CONTENTS

\$03 \$97 (STAA)

\$04 \$41 (address)

Store the contents of the accumulator A into memory location \$41.

- D. SWI (Software Interrupt):** The program counter is incremented by one. The program counter, index register, accumulators A and B, and the condition code are all pushed onto the stack. (See page 67 of the M6800 PROGRAMMING MANUAL)

Example: MEMORY ADDRESS MEMORY CONTENTS

\$05 \$3F

RESET the System X

The ASCI system uses the SWI in the ASCI T68 bug as the break point. So, any time the 3F or SWI instruction is used, the display will show the Program Counter (PC) and the Conditional Code Register (CCR).

Example: MEMORY ADDRESS MEMORY CONTENTS
 0060 bc

As you develop future programming techniques and debugging, the advantage of this display and the break point will be clear.

- E. NEGA** (Negate Accumulator or memory): This instruction negates the specified accumulator or memory location by replacing the accumulator or memory location with the 2's complement of its contents. (See page A49 of the M6800 PROGRAMMING MANUAL.)

Example: MEMORY ADDRESS MEMORY CONTENTS
 \$02 \$40 (NEGA)

Suppose the contents of the accumulator contain \$2B
 or

0010 1011. Upon this instruction, it would change to
 1101 0100 1's complement

_____ plus 1

1101 0101 2's complement \$d5 would now be in
 the accumulator.

IV. Laboratory Exercise

- A.** As an introduction to beginning machine and assembler language programs, we will develop some simple programs that will also improve your ability in the handling of numbering systems. In these programs, and in future laboratory exercises, we will use portions of the "Hello Can I Help You" program found in Chapter 3, pages 22-23. The first program will take the

instruction found at memory location \$008F, complement that number, and store the result, or written in assembler.

```
LDAA#$4A
COMA
STAA$96
SWI
```

Look on page 34 at the instructions at location \$8F and \$91. All we are doing is inserting the COMA between these two instructions. Let us now analyze each instruction.

1. LDAA#\$4A, will load the A accumulator with the value of \$4A. Since this is in the immediate mode, the value will be loaded into memory.
2. COMA will change all 1's to 0's and all 0's to 1's of the value that is in the A accumulator.
3. STAA \$96 will take the contents now in the A accumulator, and store it in memory location \$0096, but will leave that value in the A accumulator.
4. SWI will halt the program mode and display the contents of the program counter and the conditional code register.
5. To write the program we will need to use an ASCII repertoire card for coding instructions. Let's take the first instruction:

```
LDAA#$4A
```

This is, load the A accumulator immediately (# sign indicates immediately) with hexadecimal value (\$ indicates hexadecimal), so write the value on a programming sheet, just as it appears on page .

LOCATION	HEX	INSTR.	OPERAND	COMMENT
008F	86	LDAA	#\$4A	Load Display
0090	4A			

Look up LDAA immediate on the repertoire card. Notice it is the instruction 86, and the next memory location will contain the value to be loaded on \$4A.

6. The COMA is an implied instruction. Look up COMA on the repertoire card. Notice it is 43.
7. STAA\$96 — look up the code for STAA. Notice there is no immediate mode for STAA, since all contents must be loaded to a memory location, so the code for STAA will be direct mode or 97.
8. The code for SWI will be 3F.
9. Final coding of the program should look like this:

MEMORY ADDRESS	MEMORY CONTENTS	INSTRUCTION
008F	86	LDAA
0090	4A	
0091	43	COMA
0092	97	STAA
0093	96	
0094	3F	SWI

10. Enter the program as follows:
 - a. RESET the System X.
 - b. Depress and release any key for keyboard control.
 - c. Depress the "A" key for auto load mode.
 - d. Enter starting address \$008F.
 - e. Enter 86, 4A, 43, 97, 96, 3F, in that order.
 - f. RESET the System X.
11. Let's run the program and see what happens.
 - a. Depress and release any key for keyboard control.
 - b. Depress "SET PC" or "7" key.
 - c. Enter starting address 008F.
 - d. Depress the "DO" or "D" key.
 - e. Program will run and display program counter and conditional code register.

- f. To find results, depress "EXAM" or "E" key and depress the address where we stored data or \$0096. This should read \$b5. Let's see why.

Take \$4A to binary equal.	01001010
Complement 1's	10110101
Converted to HEX gives	b 5

- 12.** Change the value of memory location \$0090 to \$00 and run the program. Again, change to \$FF and run again. What are your results? Do you understand what your program is doing? If not, review until you do.
- B.** Using direct mode of addressing, modify the preceding program so the instruction at \$008F is in the direct mode rather than the immediate mode, but first let's study the difference. In the immediate mode the value in the operand is loaded in the accumulator. In the direct mode the value in the address located in the operand will be loaded into the accumulator.

Example:	MEMORY ADDRESS	MEMORY CONTENTS
LDA#\$4A or	008F	86 - Im
	90	4A

This means, the value \$4A is loaded into the accumulator.

LDA\$95 or	008F	96 - Direct
	0090	95

This means, load the value that is located in memory address \$0095 in the A accumulator. Now if the value \$4A was placed in \$0095, the program would do exactly the same as the preceding program. The question must be asked at this point, What is the difference? Please refer to your repertoire card, look at the instruction LDA, and notice under the immediate mode under the symbol $\sim$ the value of 2. This indicates it takes 2 cycles to complete this instruction. Under the direct mode, it takes 3

cycles to complete, so with the System X operating with a 1 Mhz clock, that is a time savings of 1 microsecond. That does not sound like much, but time saving is very important to the micro-computer programmer.

1. Enter the program and run it. Don't forget to RESET and then depress any key for keyboard control and ASCII UP.

MEMORY ADDRESS	MEMORY CONTENTS	INSTRUCTION
008F	96	LDAA\$95
0090	95	
0091	43 - 40	COMA
0092	97	STAA\$96
0093	96	
0094	3F	
0095	4A	DATA
0096		STORAGE

Change the value of address \$0095 to \$00. What is the result stored in address \$0096? Change the data to \$FF. What is the stored result?

C. Using the NEGA or 2's complement: By using the preceding program, and changing the instruction located at memory address \$0091 from \$43 (COMA) to \$40 (NEGA), we accomplish the 2's complement. Remember, 2's complement is 1's complement and then add 1.

1. Run the preceding program and 2's complement the value. Notice when your 2's complement \$00 you get \$00 or 2's complement \$FF and you get \$01.

Example:

\$00 = binary	0000	0000
1's complement	1111	1111
add 1		1
	1 0000	0000

Giving Carry-out Bit

2. Try some different values in memory location \$0095, and check your results by changing the HEX answer to binary, and completing binary arithmetic.
- D.** Since the 6800 MPU uses two accumulators, you should become familiar with both. Rewrite all the preceding programs using the B. accumulator. All instructions may be found on the REP CARD (repertoire card).

Chapter 6: Introduction To Program Debugging

Objective: To familiarize the student with basic techniques for program debugging through the use of break points.

Student study requirements: Pages 77, 78, 88, and 89 of BASIC MICROPROCESSORS and THE 6800 by Bishop.

Discussion

I. Debugging

- A.** The only purpose of a microcomputer is to make decisions. In order to accomplish this decision making process, the system will use status registers or in the case of the M6800, it is called the Conditional Code Register (CCR). The CCR will be covered in greater detail in Chapter 9 of this manual. Since the purpose of this chapter is to introduce you to some basic techniques in debugging using the ASCI System X keyboard, we will use the CCR and must cover an introduction into this register.
- B.** The M6800 CCR is one word long (8-bits) and holds 1-bit indicators (flags) that represent the state of conditions inside the microprocessor unit (MPU) on occasionally external inputs. The M6800 CCR maintains five status flags and an interrupt control bit. The five status flags are: Carry (C), Overflow (V), Sign (N), Zero (Z), and Auxiliary Carry (H).
- C.** The ASCI System X has five keys that will be helpful to you for programming debugging. These keys are Single Step (SS), Program Counter and CCR display (PC/CC), Index Register and A accumulator display (X/A), Stack Pointer and B accumulator display (SP/B), and set Break Point (BP). In addition, as mentioned in Chapter 5, the 3F or SWI instruction at the end of a program will automatically display the program counter and the CCR.

D. With this information to help you with debugging, follow the sample program. We realize at this point you **will not** be familiar with all the instructions in this program, but they will be covered in later chapters. Your objective now is familiarization with the tools you have available for the debugging of programs.

II. Laboratory Exercise (sample of program debugging)

A. Load the following program:

Location	Hex	Instr.	Operand #	X	Comment
0 0 6 0	4 F	C L R A			SET A to 00
0 0 6 1	5 F	C L R B			SET B to 00
0 0 6 2	C E	L D X	\$ 0 1 0 1		LOAD X WITH \$0101
0 0 6 3	0 1				
0 0 6 4	0 1				
0 0 6 5	8 6	L D A A #	0 0		LOAD A WITH 00
0 0 6 6	0 0				
0 0 6 7	0 8	I N X			ADD 1 to X
0 0 6 8	8 b	A D D A #	\$ F F		ADD FF to A
0 0 6 9	F F				
0 0 6 A	0 8	I N X			ADD 1 to X
0 0 6 B	C 6	L D A B #	\$ 2 F		LOAD B WITH \$2F
0 0 6 C	2 F				
0 0 6 D	0 8	I N X			ADD 1 to X
0 0 6 E	C b	A D D B #	\$ 5 1		ADD \$51 to B
0 0 6 F	5 1				
0 0 7 0	0 8	I N X			ADD 1 to X
0 0 7 1	3 F	S W I			STOP

Be sure to verify the program is properly loaded by use of the examine routine.

- B.** Execute the program by depressing the "SET PC" key and enter 0060 (program beginning location), then depress the "SS" (Single Step) key. The System X is now under single step control and will continue to execute this mode unless the "D" key is depressed, which will put the system under automatic mode. This is a do nothing program and is being used only to demonstrate debugging ability.

- C.** Now analyze the results of the display.

- 1.** Once you depress the "SS" key you will receive an automatic display of the program counter and the CCR which will now show 0061 (PC) F4 (CCR).

Note: The CCR may have the value d4 in at this time if you have previously run the program with no interrupt service routine. So, if you have d4 instead of F4, or d0 instead of F0, it will be correct. In later chapters, you will understand why this condition can happen.

- 2.** Depress the "X/A" key. The display will now show XXXX (Index Register) and 00 (A accumulator).

Note: The index register may have any value, but it makes no difference at this point.

- 3.** Depress the "SS" key. Display will show 0062 (PC) F4 (CCR).
4. Depress the "SP/B" key. The display will show 000b (Stack Pointer) 00 (B accumulator).
5. Depress "SS" key. The display will show 0065 (PC) F0 (CCR).
6. Depress the "X/A" key. The display will show 0101 (Index Register) 00 (A accumulator). The index register was loaded with \$0101 on this instruction.

- 7.** Depress the "SS" key. The display will now show 0067 (PC) ⁴F4 (CCR).

- a. Break the CCR value to binary:

—	—	H	I	N	Z	V	C
1	1	1	1	0	1	0	0
B7	B6	B5	B4	B3	B2	B1	B0

- b. Analyze the binary:
- (1) B0 is the carry-borrow which we do not have with 00 loaded into A accumulator.
 - (2) B1 is the overflow which is not true with 00 loaded into A accumulator.
 - (3) B2 is the zero bit which is 1 set with 00 loaded into A accumulator.
 - (4) B3 is the negative bit which is not true with 00 loaded into A accumulator.
 - (5) B4 is the interrupt bit.
 - (6) B5 is the half-carry bit.
 - (7) B6 and B7 are always one set.
- 8.** Depress the "X/A" key. Display will show 0101 (Index Register) 00(A). The index register was loaded with 0101 on the instruction in 0062. The A accumulator was loaded with 00 on this instruction.
- 9.** Depress the "SP/B" key. This will not change since we have not changed to data.
- 10.** Depress the "SS" key. Display will show 0068 (PC) F0 (CCR). the PC has incremented; the CCR has not worked with any value, so resets to 0, except for the interrupt bit which will always be 1 in single step mode. Single step mode uses the 3F or SWI which is an interrupt.
- 11.** Depress the "X/A" key. Display will show 0102 (Index Register) 00(A). This instruction increments the index register. Accumulator A was not affected.
- 12.** Depress the "SP/B" key. No change.
- 13.** Depress the "SS" key. Display will show 006A (PC) d8(CCR). The PC was incremented. Since the A accumulator was loaded with FF, which is a negative value, the negative indicator went one set, bit-3.

14. Depress the "X/A" key. Display will show 0102 (Index Register) FF (A). The index register had no change, but we added \$FF to \$00 in accumulator A, giving a result in accumulator A of \$FF.
15. Depress the "SP/B" key. No change.
16. Depress the "SS" key. Display will show 006b (PC) d8 (CCR). No change. The instruction was to INX which will have no effect on accumulators A or B or the CCR - only the index register.
17. Depress the "X/A" key. Display will show 0103 (Index Register) FF (A). The index register was incremented, but no effect on accumulator A.
18. Depress the "SP/B" key. No change.
Note: It is worthwhile at this point to mention that you may go into the examine mode while in the single step mode. Depress the "E" key at this point and address it with 0067, which will contain the value \$08. You may change, go backward ("B" key), forward ("F" key), and address any location. In other words, while in a program under single step mode, you can change that program and return to the original program by depressing the "SS" key.
19. Depress the "SS" key. Display will show 006d (PC) d0 (CCR). PC is incremented. The CCR changed since the value \$2F was loaded into the B accumulator, and all bits will reset except the interrupt bit.
20. Depress the "X/A" key. No change. We did not affect the index register or the A accumulator.
21. Depress the "SP/B" key. Display will show 000b (SP) 2F (B). The instruction called for \$2F to be loaded into the B accumulator.

22. Depress the "SS" key. Display will show 006E (PC) do (CCR). The PC was incremented, but no change to the CCR.
23. Depress the "X/A" key. Display will show 0104 (Index Register) FF (A). The instruction incremented the index register, but no change to the A accumulator.
24. Depress the "SP/B" key. Display will show no change.
25. Depress the "SS" key. Display will show 0070 (PC) FA ^{CCR} (A). The program counter was incremented and the CCR was reset by the addition of \$2F and \$51, giving a result of \$80. This will cause an overflow since the number is greater than \$7F. It will also set the negative bit. Since bit-7 of the B accumulator is one set, the number will be negative. The half-carry will go set since we have a binary carry.

Example:

0010	1111	\$2F
0101	0001	\$51
<u> </u>		
Add	1000	0000 \$80

26. Depress "X/A" key. Display will not change. No effect.
27. Depress "SP/B" key. Display will show 000b (SP) 80 (B). The B accumulator will show the result of adding \$2F and \$51.
28. Depress the "SS" key. Display will show 0071 (PC) FA ^{CCR} (A). PC will increment. No change to CCR.
29. Depress the "X/A" key. Display will show 0105 (Index Register) FF (A). The index register was incremented by the instruction.
30. Depress "SP/B" key. Display will not change.
31. Depress "SS" key. Display will show 0072 (PC) FA ^{CCR} (A). The instruction was an SWI which calls for display of the PC and CCR.

D. Break Point/Debugging

- 1.** The Break Point key (BP) may also be used for debugging. The program you have just completed should be left in the system.
- 2.** RESET the system, depress and release any key.
- 3.** Depress "SET PC" key. Load 0060.
- 4.** Depress "BP" key. Load 0066. The address 0068 is the location we are using to insert a Break Point (BP). Any address in the program containing an instruction may be used. (See the explanation of the break point key in the USER'S MANUAL.)

Example: Display will show 0068 (PC) do (CCR)

Depress "X/A" 0102 (X) 00 (A)

Depress "SP/B" 000b (SP) 00 (B)

- 5.** You may use the BP insertion in any program. To return from a BP, depress either the "DO" or "SS" key — which ever you desire.

Page 44

Add the following paragraph to this page:

- 6.** Please note: The breakpoint (BP KEY) may only be used on programs stored in RAM. You cannot use the single step mode or set a breakpoint on programs stored in ROM.

Chapter 7: Introduction to Input/Output

Objective: To familiarize the student with memory mapped I/O; conditional branch instructions; masking; conditional codes; extended, immediate, and relative addressing; seven-segment displays; and delay programming.

Student study requirements: Chapter 7 of BASIC MICROPROCESSORS and THE 6800 by Bishop; and pages A6, A10, A19, A20, A42, A47, A48, and A66 of the M6800 PROGRAMMING MANUAL.

Discussion

I. New Addressing Modes

- A.** Immediate Addressing Mode: The addressing mode in which the data itself is part of the instruction and is found in the next memory location.

Example:	MEMORY ADDRESS	MEMORY CONTENTS
	\$0000	\$86 (LDAA immediate opcode)
	\$0001	\$5F (data)

\$86 is the LDAA immediate opcode, and \$5F is the data. The result after execution is that the \$5F has been loaded into accumulator A.

Note: The symbol # signifies the immediate mode of addressing. The above written mnemonic would be LDAA#\$5F.

- B.** Extended Addressing Mode: This mode of addressing is used to address memory locations above 00FF (\$FF = 255_{10} so $256_{10} = \$0100$).

The second memory byte contains the most significant bits of the memory location, while the third memory byte contains the least significant bits of the memory location.

Example:	MEMORY ADDRESS	MEMORY CONTENTS
	\$0000	\$B6 (LDAA extended opcode)
	\$0001	\$03 (address of high byte)
	\$0002	\$FF (address of low byte)

\$B6 is the LDAA extended opcode. \$03 is the most significant half of the address, and \$FF is the least significant half of the address where the data is stored. After execution, the contents of \$03FF would be loaded into accumulator A.

1. To add two NBRs together that are greater than \$FF (255), you must use additional locations (no storage, any memory location, can store ^{no} more than two digits — 00 to FF).

2. Try a program using extended mode.

Example: Location 0101 (instead of 0050). Recall that 0050 is simply 50 - 00 not necessary.

0050	B6	LDAA Extended with memory location
	01	(high byte)
	01	(low byte)
	B7	Store A reg. (extended) in memory
	02	
	02	
	3F	

DATA	
01	A
01	A

Example: Transfer **forward** from the present location. Assume it is desired to branch from the present location \$A100 to \$A147. First it should be verified that the branch is not beyond the allowable range of positive 129 locations from the present location. A150 +2 A150

\$45 is within our allowable range. The 8-bit, 2's complement of the number \$45 is the number itself \$45 (then reverse will give \$45).

$$PC \leftarrow (PC) + 0002 + REL$$

A147 to A102
 Not more than 128 $\longrightarrow$ $\begin{array}{r} \$A147 \\ A102 \\ \hline \$45 \end{array}$

$\$40 = 64$
 $\$05 \quad \underline{05}$
 $\$45 = 69_{10}$

$\begin{array}{ccccccccc} & 4 & & & 5 & & & & \\ \boxed{0} & \boxed{1} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{1} & \boxed{0} & \boxed{1} & \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & \end{array}$
 $2^6 = 69_{10}$

128 x 8 bits = 1024, so user has
 128 bytes

$\$45$ 01000101₂
 1's comp. 10111010₂
 2's comp. 1

MSB is the sign bit, so actually
 becomes 0011 1011 or 3B

$10111011 = \$BB$

Note: If the sign bit is 0, the NBR is positive.

0011 1011 = +3B

Sub one $\underline{-1}$ (reverse of 2's compliment)

Compliment 0011 1010
 1100 0101 = -45

(See page 44 of BASIC MICRO-
 PROCESSORS and THE 6800
 by Bishop).

or
 $\begin{array}{r} 1011 \ 1011 \\ \underline{-1} \\ 1011 \ 1010 \end{array}$
 Compliment 0100 0101 = 45

MEMORY ADDRESS	MEMORY CONTENTS
$\$A100$	$\$20$ (BRA opcode)
$\$A101$	$\$45$ (offset)
$\$A102$	XX (present value of the program counter)
$\$A147$	XX (opcode for next instruction)

$\$20$ is the BRA (Branch Always Opcode). $\$45$ is the offset or number of locations which will be branched over starting with $\$A102$. Therefore, the next instruction the MPU will execute will be located at A102 plus $\$45$, or location $\$A147$.

Example: Transfer **back** from the present location. Assume it is desired to branch from the present location of A100 back to memory location A090. This is accomplished in a similar manner as the forward branch except the number of locations is a **negative** number expressed

in 2's complement from the present location plus 2. The 2's complement form of negative number places 1 in bit-7MSB which, in effect, tells the MPU TO BRANCH BACKWARD RATHER THAN FORWARD.

Present Location plus 2	\$A102	102 = 258	FF = 255
		090 = 144	102 = 258 ₁₀
Final location	<u>\$A090</u>	= 114 ₁₀	090 = 144 ₁₀
			\$72 = 114 ₁₀
Number of locations back	\$72		

TO REPRESENT - 72 in 2's complement, first write the binary representation of \$72

\$72 equals %0111 0010
 1's complement %1000 1101
 add 1 %1000 1110
 -\$72 equals \$8E(2's complement)

<u>MEMORY LOCATION</u>	<u>MEMORY CONTENTS</u>
\$A090	XX (opcode of next instruction after branch)
•	•
•	•
\$A100	\$20 (BRA opcode)
\$A101	\$90 (offset)
\$A102	XX (present value of the program counter)

\$20 is the BRA (BRANCH ALWAYS opcode) \$90 is the offset or number of locations which will be branched back over starting from A102. Therefore, the next instruction the MPU will execute will be located at memory location \$A090.

II. New Instructions

- A. AND (AND Memory with Accumulator):** This instruction ANDs the contents of a memory location with the contents of the accumulator. (See page A6 of the M6800 PROGRAMMING MANUAL.)

Note: The symbol % signifies binary.

Example:	MEMORY ADDRESS	MEMORY CONTENTS
	\$03	\$84 (opcode for ANDA)
	\$04	\$20

This program uses the opcode for ANDA immediate with memory location \$04. Written in mnemonic, this would be ANDA #%0010 0000, or what is the accumulator with \$20.

- B. BCS (Branch if Carry Set, opcode \$25):** This instruction will execute only if the carry bit is set, otherwise the next instruction is executed. (See page A10 of the M6800 PROGRAMMING MANUAL.)
- C. BMI (Branch if Minus opcode \$2B):** This instruction will execute only if the sign bit is a 1, otherwise the next instruction is executed. (See page A19 of the M6800 PROGRAMMING MANUAL.)
- D. BNE (Branch if Not Equal, opcode \$26):** This instruction will be executed only if the zero status is 0, otherwise the next instruction will be executed. (See page 20 of the M6800 PROGRAMMING MANUAL.)

Note: All branching instructions follow the format given under relative addressing modes covered earlier in this chapter. The BRA instruction is the beginning of all branches, so once the student becomes familiar with that instruction, all other branching instructions are operated the same, only the status code will change.

- E. INX (Increment Index Register, opcode \$08):** The instruction will add 1 to the 16-bit value in the index register. (See page A42 of the M6800 PROGRAMMING MANUAL.)

Example: Suppose the index register contains \$0100. After the instruction INX, the index register will contain \$0101. This instruction will be used considerably for delay routines.

- F. LDX (Load Index Register):** Load the contents of two selected memory locations into the index register. This instruction loads the contents of the selected memory byte into the higher byte of the index register. Load the contents of the memory byte immediately following the selected memory byte into the low byte of the index register. (See page A47 of the M6800 PROGRAMMING MANUAL.)

Example:	MEMORY ADDRESS	MEMORY CONTENTS
	\$0100	\$FE (opcode for LDX extended)
	\$0101	\$03
	\$0102	\$FF
	\$03FF	\$01
	\$0400	\$00

This instruction LDX takes the value of locations \$03FF and \$0400 and places that into the index register, or in this case, \$0100 would be loaded into the index register.

- G. LSR (Logical Shift Right):** This instruction performs a 1-bit logical right shift of the specified accumulator or the selected memory location. (See page A48 of the M6800 PROGRAMMING MANUAL.)

Example: Suppose accumulator B holds \$7A and the instruction \$54 is given (LSRB). The result would be \$3D in the B accumulator.

\$7A equals % 0111 1010

Shifted right one bit equals % 0011 1101

Giving the result of \$3D

H. SUB (Subtract): Subtract the contents of the selected memory byte from the contents of the accumulator A or B. (See page A66 of the M6800 PROGRAMMING MANUAL.)

Example:

MEMORY ADDRESS	MEMORY CONTENTS
0100	\$B0 (SUBA)
0101	\$01
0102	\$10
0110	\$A0

Suppose the accumulator had \$E3, the value found in location \$0110, which is \$A0.

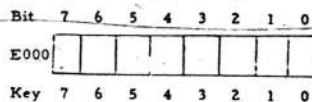
\$E3 equals	%1110 0011
\$A0 2's complement	<u>\$0110 0000</u>
SUB result	\$0100 0011

Remember, subtraction is carried out by the use of 2's complements as covered in Chapter 2.

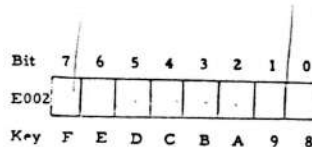
III. Laboratory Exercise

A. ASCII System X Input/Output

1. The System X treats input/output devices just like memory locations. For example, on the System X, the 16-key keyboard uses memory addresses E000 and E002. Memory location E000 has one bit for each of keys "0" through "7" as follows:



Memory location E002 similarly has one bit for each of keys "8" through "F":



The bit is 0 if the key is being pressed, 1 otherwise.

2. Although you may use memory locations E000 and E002 just like any other locations, it does not make much sense because it may be lost.

Note: Since these addresses use E000 and E002, you will have to use the extended mode. This is the fourth addressing mode on the instruction card.

B. An Electronic Lock

1. The program in B. 2. will blank out the display until you press a particular key, and then will revert control to the monitor. The key to revert to monitor is key "6".

Note: The key will revert to the monitor routine and display that particular command, (e.g., key "6" is SP/B display. The student should look at each instruction.

- a. LDAA \$E000 loads accumulator A with the content of memory location E000 (HEX), i.e., with the state of keys "0" through "7". LDAA extended is B6, not 86!
- b. ANDA #00100000 logically ANDs the contents of accumulator A with the binary number 00100000. The # means "immediate" or, the data is in the next word rather than somewhere in memory. The 0 means "binary." The result of the logical AND is 0 if key "6" is being pressed (bits = 0), and 01000000 if switch 6 is not being pressed. This process is called **masking** and will be discussed in detail in a subsequent lesson.
- c. BNE causes the computer to branch if the zero bit is 0. Otherwise, the computer goes to the next instruction in sequence, i.e., SWI, (the software interrupt).

2. The program in HEX is as follows:

Location	Hex	Instr.	Operand # , X	Comment
0 0 6 0	B 6	L D A A	E 0 0 0	WAIT
0 0 6 1	E 0			
0 0 6 2	0 0			
0 0 6 3	8 4	A N D A	# 4 0	%01000000
0 0 6 4	4 0			01000000
0 0 6 5	2 6	B N E		GO TO WAIT
0 0 6 6	F 9			
0 0 6 7	3	F S W I		

- a. LDAA \$E000 uses the extended mode and has the address in the next two words. It should be noted that the most significant digits are first.
- b. ANDA #01000000 uses the immediate mode and has the data in the next word. Masking is seen better if in binary.
- c. BNE WAIT uses a relative address with distance by subtracting the address of the next instruction from the address of the destination. In this example the distance is:

Drop the first two zeros, so: 60 = 0110 0000
 -67 = 0110 0111

1's complement of 67 = 1001 1000	60 = 0110 0000	
2's complement is 1001 1001	so	$ \begin{array}{r} 60 = 0110\ 0000 \\ +\ 1001\ 1001 \\ \hline 1111\ 1001 \\ \text{F}\quad\text{9} \end{array} $

- d. Hexadecimal subtraction is tricky. Try working several examples. You can drop the two most significant digits. Remember that subtraction is the same as adding the 2's complement (1's complement plus 1).
 - e. Enter and run the program. What happens when you press key "0"? What happens when you press key "6"?
 - f. Change the program so it responds to key "4" and run it. Next make the program respond to key "3".
- 3.** Key "7" is a little different to use since you can test bit-7 as the negative (N) bit (bit-7 is the MSB). The following hexadecimal program should help you understand this important modification to your program:

Location	Hex	Instr.	Operand # , X	Comment
00,60	B6	LDA	A	WAIT GET KEYS "0" - "7"
00,61	E0			
00,62	00			
00,63	2B	BMI		GO TO WAIT. IS 7 NOT DEPRESSED
00,64	FB			
00,65	3F			

a. BMI causes a branch if the negative bit is 1. Remember, the negative bit is the most significant bit (bit-7) of the previous result. (Look up the instruction.)

b. In this example the offset is:

60 =	0110	0000		0110	0000
65 =	0110	0101	SO	+	1001 1011
1's comp.	1001	1010		FB =	1111 1011
2's comp.	1001	1011			

This is an area that encourages mistakes. Do it again!

c. Enter the run the program for key "7".

Note: Key "7" is Set PC and that is what you will return to.

d. Another key that is fairly simple to use is the "0" key. Try this:

```

WAITK   LSR   E000   IS KEY "0" BEING PRESSED?
        BCS   WAITK NO WAIT
        SWI

```

e. Enter and run this program. LSR extended is 74 and BSC is 25.

C. A Combination Lock

1. You can combine programs to wait for more than one key. The following program will wait for keys "2" and "5" in that order: (actually, depressing any key combination, so long as "2" and then "5" are keyed, will work).

Location	Hex	Instr	Operand # , X	Comment
0,0,6 0	B,6	L,D,A	A E,0,0,0	WAIT 1
0,0,6 1	E,0			
0,0,6 2	0,0			
0,0,6 3	8,4	A,N,D,A	#,0,4	%00000100 IS KEY "2" PRESSED?
0,0,6 4	0,4			
0,0,6 5	2,6	B,N,E		NO, GO TO WAIT 1
0,0,6 6	F,9			
0,0,6 7	B,6	L,D,A	A E,0,0,0	WAIT 2
0,0,6 8	E,0			
0,0,6 9	0,0			
0,0,6 A	8,4	A,N,D,A	#,2,0	%00100000 IS KEY "5" PRESSED? 5210
0,0,6 B	2,0			
0,0,6 C	2,6	B,N,E		NO, GO TO WAIT 2
0,0,6 D	F,9			
0,0,6 E	3,F	S,W,I		

2. Enter this program. Did it work? Again we see a branching example.

In this example, there are two.

- a. BNE Wait for first key 0060 = 0110 0000
 — 0067 = 0110 0111
 — 07 = F9

— 07 = +F9 (This example was shown in section B2C
 F9 of this experiment)

- b. BNE Wait for second key 0067 = 0110 0111 0110 0111
 006E = 0110 1110 = 1001 0011
 1111 1001
 F 9

3. Modify the last program to perform the following tasks:

- a. Wait for key "4" followed by key "8".

- b. Now try for either key "0" followed by key "7" or key "7" followed by key "0". Can you see any way to modify or improve the existing program? Hint: Try to replace ANDA instruction with SUBA.

D. Using the System X LED Display

1. The System X LED (Light Emitting Diode) displays use memory locations E402 and E400. Memory location E402 is the data, and memory location E400 determines which displays will be visible.
2. Each seven-segment display consists of seven-segment LEDs and a decimal point organized as shown in Figure 7-1. The System X uses the following arrangement to control the LEDs:
 - a. Memory location E402 has one bit for each segment and the decimal point as follows:

Bit	7	6	5	4	3	2	1	0
E402	a	b	c	d	e	f	g	dp

A 1 in the bit position turns the LED on, and a 0 turns it off, e.g., 0101 0000 would light segments b and d.

- b. Memory location E 400 has six bits which control the entire seven-segment display. The bit selection is as shown:

Bit	7	6	5	4	3	2	1	
E400	Not Used	LED = 1	LED #2	LED #3	LED #4	LED #5	LED #6	Not Used

A 1 in the bit position turns the display on. The LED display is shown.

LED #1	LED #2	LED #3	LED #4
-----------	-----------	-----------	-----------

LED #5	LED #6
-----------	-----------

LEDs 1 through 4 are the address displays (on the left), and 5 and 6 are the data displays (on the right).

E. Activating and Display

1. We will start with a program to light all the decimal point LEDs:

Location	Hex	Instr.	Operand #, X	Comment
0.0.6.0	8.6	L.D.A.A	#.F.F.	TURN DISPLAY ON START
0.0.6.1	F.F			
0.0.6.2	B.7	S.T.A.A		
0.0.6.3	E.4			
0.0.6.4	0.0			
0.0.6.5	8.6	L.D.A.A	#.0.1.	TURN DECIMAL POINT ON
0.0.6.6	0.1			
0.0.6.7	B.7	S.T.A.A		
0.0.6.8	E.4			
0.0.6.9	0.2			
0.0.6.A	3.F	S.W.I		

Note: The LDAA immediate (86) loads accumulator A with the contents of the next memory location given.

2. Enter and run the program. Did the program light the decimal points? The problem is that the program turns the displays on for a few microseconds before the System X reverts to the monitor routine. Thus, the monitor takes control and puts the program counter and the conditional code register on the displays. You can solve the problem by not returning to the monitor. To do this, end the program with the instruction 20 (recall this is a branch instruction).

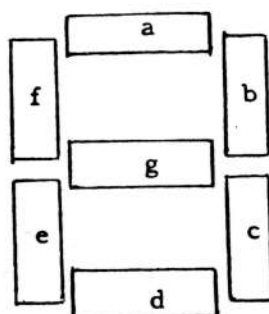
Location	Hex	Instr.	Operand # , X	Comment
0.0.6.A	2.0	.	B.R.A.	GO TO START
0.0.6.B	0.0 F4	.	.	

3. Now run the program. What happens? Table 7 - 1 shows the patterns required in location E400 to turn on the various displays: Table 7 - 2 shows the patterns required in location E402 to turn on the individual segments. Try some combinations. Remember that one is on, zero is off. An example to turn on more than one segment:

Example:

Segments	Pattern (HEX)
a and b	C0
a and e	88
b and c	60
b and g	42
e and f	0C

Figure 7 - 1
Seven-Segment
Display



4. The display: Draw a picture of one LED array.

How many segments are available?

Did you remember the decimal point????

5. Display limitations: What letters are **not** possible? List them.

6. What combination of keys must be pressed to display:

A

E

•

B

F

8

C

R

4

Table 7 - 1
Patterns for Turning on Displays
 (Memory Location E400)

Display Number	Pattern (HEX)
1	40
2	20
3	10
4	08
5	04
6	02

Bits 0 and 7 do not matter since they are not used. To turn on more than one display, just set each bit.

Example:

Display Number	Pattern (HEX)
1 and 2	E1
1 and 5	C5
2 and 3	B1
3 and 5	95
4 and 6	8B

Table 7 - 2
Patterns for Turning on Segments
 (Memory Location E402)

Segments	Pattern (HEX)
a	80
b	40 -
c	20 -
d	10 -
e	08 -
f	04 -
g	02 -
dp	01 -

F. Delaying the Return to Monitor

1. In most applications, you will want to control for a specified amount of time. An example program provides a delay by counting with the index or X register:

```

COUNT    LDX #0          DELAY LOOP
           INX
           BNE COUNT

```

Look up the LDX instruction in the M6800 PROGRAMMING MANUAL. The X register is 16-bits of 4 hexadecimal digits long so the program counts from 0000 to FFFF before it reaches 0 again. When the computer adds 1 to FFFF, the result is 0. Try it by examining memory location FFFF and then pressing F (Forward) and B (Back). Consider time elapsed.

2. Enter the delay loop at the end of the display program:

```

006A          LDX #0          CE
006B          00
006C          00
006D  COUNT  INX            08
006E          BNE COUNT      26
006F          FD
0070          SWI            3F

```

3. Run the program. Notice what has happened. You can change the length of the delay by changing memory locations 6B and 6C (not starting at 0000). Try different delay times. What is the maximum delay time?
4. You can turn on first one segment (decimal point) and then another (g) with the following long and rather difficult program:

Location	Hex	Instr.	Operand # , X	Comment
0.0.6.0	8.6	L.D.A	A # F F	TURN DISPLAY ON
0.0.6.1	F.F			
0.0.6.2	B.7	S.T.A	A	
0.0.6.3	E.4			
0.0.6.4	0.0			
0.0.6.5	8.6	L.D.A	A # 0 1	BEGIN

01							
0.0,6,6	0.0						% 0000 0001
0.0,6,7	B.7	S.T.A	A				TURN DECIMAL POINT ON
0.0,6,8	E.4						
0.0,6,9	0.2						
0.0,6,A	C.E	L.D.X					LOAD DELAY COUNTER
0.0,6,B	0.0						
0.0,6,C	0.0						
0.0,6,D	0.8	I.N.X					DELAY 2
0.0,6,E	2.6	B.N.E					GO BACK TO DELAY 2
0.0,6,F	F.D						
0.0,7,0	8.6	L.D.A	A	#.F.D.			
0.0,7,1	F.D						% 1111 1101
0.0,7,2	B.7	S.T.A	A				TURN BAR ON
0.0,7,3	E.4						
0.0,7,4	0.2						
0.0,7,5	C.E	L.D.X					LOAD DELAY COUNTER
0.0,7,6	0.0						
0.0,7,7	0.0						
0.0,7,8	0.8	I.N.X					DELAY 2
0.0,7,9	2.6	B.N.E					GO TO DELAY 2
0.0,7,A	F.D						
0.0,7,B	7.E						GO TO TURN DISPLAY ON
0.0,7,C	0.0						
0.0,7,D	6.0						
E							
F							

- Run this program. If it does not run, recheck your entry by keying forward step by step.
- What happens when you change the delay?

Chapter 8: Understanding the Data Arrays, and Other Multiple Functions

Objective: To familiarize the student with arrays, index addressing, ending markers pointers and counters, instruction execution, and program loops.

Student Study requirements: BASIC MICROPROCESSORS and THE 6800 by Bishop; pages A11, A21, A29, A36, A43, A50, and A65 of the M6800 PROGRAMMING MANUAL.

Discussion

- I. Indexed Addressing Mode: In this mode of addressing, the second byte of the instruction combines with the contents of the index register to produce the memory address to be used as the operand.

Example: Index register contains \$A014 (This has been placed in the register.)

MEMORY ADDRESS	MEMORY CONTENTS
\$0100	\$A6 LDAA indexed
\$0101	\$21 (offset)
\$A035	\$B7

\$A6 is the LDAA indexed opcode; \$21 is the offset. To the index register, the offset of \$21 is added to form a new effecting address of \$A035 ($A014 + \21). After execution of the above instruction, the contents of \$A035 will be added into accumulator A, (in the sample, \$B7 will be added into A).

Note: The symbol "X" indicates Indexed Mode (LDAA \$21, X). (See the M6800 PROGRAMMING MANUAL.)

II. New Instructions (These instructions will add flexibility to your ability.)

- A. ABA (Add accumulator B to accumulator A):** This refers to adding the contents of accumulator B to the contents of accumulator A, where the information is stored. (See page A3 of the M6800 PROGRAMMING MANUAL.) Can be addressed or used in the inherent mode with a HEX opcode of 1B. (This instruction is a good math tool.)

Example: If A contains \$92 and B contains \$3A, after instruction ABA, A will contain \$CC.

A %	1001 0010	\$92
B %	<u>0011 1010</u>	<u>\$3A</u>
A = %	1100 1100	\$CC

- B. ASL (Arithmetic Shift Left):** This performs a 1-bit arithmetic left shift of either accumulator on the selected memory location. (See page A7 of the M6800 PROGRAMMING MANUAL.)

Example:

MEMORY ADDRESS	MEMORY CONTENTS
\$0100	\$86 LDAA
\$0101	\$44
\$0102	\$48 ASLA

The \$44 would be loaded into A, then shifted left changing the value to 88 and left in the accumulator A.

\$44 into A	01000100	\$44
ASL into A	10001000	\$88

Note: ASL is often used in multiplication. A single ASL will multiply its operand by two. This is another math tool. You should also be aware of the carry zero, and overflow status bits. (This will be expanded later, but you should realize that bits may be lost in shifting.)

- C. BPL (Branch if Plus):** This is the instruction that will be executed if the sign status is zero or plus, otherwise the next instruction will be executed. (See page A21 of the M6800 PROGRAMMING MANUAL.) The BPL instruction and offset operate the same as other branching instructions covered in Chapter 7.
- D. JMP (Jump):** The program will jump to the instruction specified by the operand by loading the address of the selected memory location into the program counter. (See page A43 of the M6800 PROGRAMMING MANUAL.)

Example:

MEMORY ADDRESS	MEMORY CONTENTS
\$0020	\$7E JMP
\$0021	\$01
\$0022	\$00
•	•
•	•
\$0100	\$86 LDAA

The program would execute a jump from \$0020 to \$0100 which contains an LDAA, which is the next operand. (This example uses extended mode.)

Note: The JMP instruction can be useful when programming to jump over locations where program may be added later.

- E. NOP (No Operation):** A single byte instruction which performs no operation except to increment the program counter. NOP uses the inherent addressing mode, and the machine language code is \$01. The NOP is useful for labels in assembler language and also to add time to delays as it requires two cycles to complete. (See page A50 of the M6800 PROGRAMMING MANUAL, and page 116 of BASIC MICROPROCESSORS and THE 6800 by Bishop.)

- F. DEC (Decrement):** This instruction subtracts one from the contents of either accumulator or memory, (page A36 of the M6800 PROGRAMMING MANUAL).

Example:

MEMORY ADDRESS	MEMORY CONTENTS
\$0200	\$86 LDAA, \$3A
\$0201	\$3A 0011 1010
\$0203	\$4A DEC \$3A
\$0204	\$B7 STAA \$39 in 02A0
\$0205	\$02
\$0206	\$A0

This program would load \$3A to accumulator A decrement to \$39 (0011 1001) and store to location \$02A0. (**Note:** 0011 1010 less 1 = 0011 1001)

- G. STX (Store Index Register):** An instruction that stores the contents of the most significant byte of the index register in the memory location as specified in the program, and stores the least significant byte of the index register at the next memory location. (See page A65 of the M6800 PROGRAMMING MANUAL.) Remember that the index register is two bytes long.

Example: Assume the contents of index register contains \$0200.

MEMORY ADDRESS	MEMORY CONTENTS
\$0100	\$FF STX (extended)
\$0101	\$20
\$0102	\$10

After execution, location \$2010 will contain \$02, and location \$2011 will contain 00. (Try this again with different values.)

- H. CLR (Clear):** An instruction that replaces the specified accumulator or memory location with all zeros. (See page A29 of the M6800 PROGRAMMING MANUAL.)

- I. BEQ (Branch if Equal): An instruction that follows the other branching instructions as previously covered. The instruction will branch if equal or zero status equals 1, otherwise the next instruction is executed. BEQ uses the relative address mode with an instructional code of \$27. (See page A11 of the M6800 PROGRAMMING MANUAL.)
- J. EOR (Exclusive OR): This performs the exclusive OR logical function between the contents of the specified accumulator and selected memory location. (See page A39 of the M6800 PROGRAMMING MANUAL.)

Example: Assume the A accumulator contains \$A4.

MEMORY ADDRESS	MEMORY CONTENTS
\$0100	\$88
\$0101	\$1F
\$A4 equals %10100100	
\$1F equals %00011111	
EOR	11000011 1011 1011

After the above instruction is executed, ~~\$C3~~ will be found in the A accumulator. *\$ b.b*

- K. CMP (Compare): This subtracts the contents of the specified memory from the specified accumulator, affecting the flags, but does not store the difference in the accumulator. The difference is forgotten and original data remains in the accumulator, and a

decision can be made from the status register. This instruction is most frequently used to set statuses before execution of branch instructions. (See page A31 of the M6800 PROGRAMMING MANUAL.)

Example:

MEMORY ADDRESS	MEMORY CONTENTS
\$0100	\$B1 CMP \$0120
\$0101	\$01
\$0102	\$20
•	•
•	•
•	•
\$0120	\$16

1. Suppose the accumulator A contains \$D3. The instruction states to compare \$16 which is located in \$120 to accumulator A which contains \$D3.

The binary statement reads: \$D3 = % 11010011
 2's complement of \$16 = % 11101010

10111101



- a. Set C to 0 since C is the complement of resulting carry.
- b. Set N to 1.
- c. Set V to 0 since $1 \times 1 = 0$
- d. Non-zero result, set Z to 0.

	H	1	N	Z	V	C
CCR			1	0	0	0

Note: The student should review the Status Register. (See page 194 of BASIC MICROPROCESSORS and THE 6800 by Bishop.)

2. The System X has a status register which maintains five status flags and an interrupt control bit. The five status flags are:

- a. Carry (C)
- b. Overflow (V)
- c. Sign (N)
- d. Zero (Z)
- e. Auxiliary Carry (H)

3. Status are assigned bit positions within the status register as follows:

	7	6	5	4	3	2	1	0	Bit No.
CCR	1	1	H	I	N	Z	V	C	

Bit numbers 6 and 7 are unassigned and permanently set to 1.

4. I is the external interrupt enable/disable flag. When it is 1, interrupts via IRQ are disabled; when it is 0, interrupts via IRQ are enabled.
5. Most 6800 literature refers to the sign bit as the symbol N, and the overflow bit as the symbol V. The immediate carry bit represents the standard carryout of bit-3, and referred to as the half carry bit, given the symbol H.
- L. DEX (Decrement Index register): This subtracts 1 from the 16-bit value in the index register. Zero status is set to 1 if the 16-bit in the result is zero. (See page A38 of the M6800 PROGRAMMING MANUAL.)

Example:

MEMORY ADDRESS	MEMORY CONTENTS
\$0100	\$09 DEX

Suppose the index register contained \$02AA. After instruction, the value \$02A9 will be found in the index register.

III. Laboratory Exercise

A. Handling Data Arrays on the System X

Some tasks involve applying the same computer instructions to an entire set of data called an array. Examples of such tasks might be calculating averages, or collecting data for storage on magnetic tape. You might even arrange a sequence of operations for an industrial control system. You may think of many more. Sequence operation is a very big field for the processor.

B. Indexing

The System X uses indexing as a simple way to apply the same instructions to each element in an array of data. Indexing means that the processor adds the address in the index register to the "offset" (or the next word following the instruction) to get the actual address of the data. The actual address used by the instruction is called the "effective address." (Understand this.)

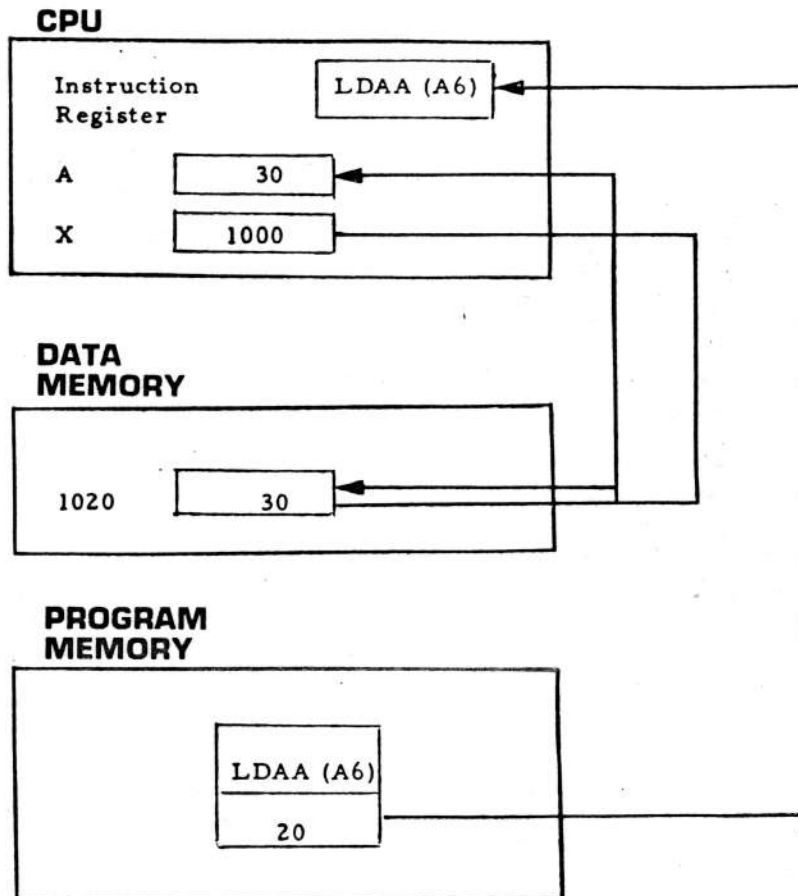
Example: The instruction LDAA \$20, X works as follows (Figure 8 -1):
Note: LDAA indexed, the processor adds \$20 to the contents of the index register, in this case \$1000. The result is \$1020. The processor places the contents of memory location 1020 in accumulator A. (Whatever was actually in memory location 1020.)

Note: The index register is a 16-bit (4-HEX digits) long while the offset is only 8-bits (2-digits) long. You should see by this example that you can change the address from which the CPU gets the data by simply changing the contents of the index register. This is a very functional addition to your growing ability to program the System X.

Example: INX (08) adds 1 to the index register and DEX (09) subtracts 1 from it. Think how easy it is to go through a series of locations. (These instructions simply change the index register by one.)

Figure 8 - 1

Execution of the LDAA \$20, X Instructions



Example:

We can change the location in data memory by changing the value in the index register. If for example, the CPU executes an INX instruction, the next LDAA \$20, X will fetch the data from address 1021. This instruction leads us to a sum of data.

C. Sum of Data

1. The following program will add the contents of memory locations 81, 82, 83, and 84, and place the sum (ignoring carries) in memory location 80. (You may use any locations available to you from the memory map.)

Note: This is a rather complicated program, so review it!

Location	Hex	Instr.	Operand #	X	Comment
0,0,6 ⁰	C,6	L,D,A	B,4		LOAD B ACCUMULATOR WITH 04
0,0,6 ¹	0,4				
0,0,6 ²	4,F	C,L,R	A		CLEAR A ACCUMULATOR
0,0,6 ³	C,E	L,D,X			LOAD INDEX REGISTER WITH 0081
0,0,6 ⁴	0,0				
0,0,6 ⁵	8,1				
0,0,6 ⁶	A,B	A,D,D	A	0,0,X	ADD CONTENTS OF A TO CONTENTS
0,0,6 ⁷	0,0				OF MEMORY LOCATION 00
0,0,6 ⁸	0,8	I,N,X			INCREMENT INDEX REGISTER
0,0,6 ⁹	5,A	D,E,C	B		DECREMENT B ACCUMULATOR
0,0,6 ^A	2,6	B,N,E			BRANCH IF NOT ZERO
* 0,0,6 ^B	F,A				
0,0,6 ^C	9,7	S,T,A	A		STORE SUM IN LOCATION 80
0,0,6 ^D	8,0				
0,0,6 ^E	3,F	S,W,I			

Note: We are using the indexed form of ADDA.

2. Try running this program with the following array of data:

(81) = 07

(82) = 23

(83) = 31

(84) = 20

The result should be:

(80) = 7B

Remember that all the numbers are hexadecimal (not decimal).

3. Now replace (82) with F1. What is the result? Can you explain this?

4. Change the program so that it will handle any array, or try this array of numbers:

* Notice the branch instruction. Do you agree? Return to exercise #7 and review if you do not agree with FA.

Note: Each location must be "loaded" first.

Example: \$07 in location (81) 07 (84) 20
 (82) 23 (85) 16
 (83) 31 (86) 38

The result should be C9 in location 80. Did you get C9? You should check this. **Hint:** Change HEX to decimal, then add.

D. Ending an Array by Using an Ending Marker

1. Sometimes you are not sure how long your array of data might be (or do not want to bother counting it), you can always end the array with a special marker. Zero makes a good marker since it adds nothing to the sum. An example program will help:

Location	Hex	Instr.	Operand # , X	Comment
0,0,6,0	4F	C.L.R.A		CLEAR A
0,0,6,1	C.E	L.D.X	#,0,0,8,1	LOAD INDEX REGISTER WITH 00
0,0,6,2	0,0			00
0,0,6,3	8,1			81
0,0,6,4	E,6	L.D.A.B	0,0,X	ADD LOAD B REGISTER WITH
0,0,6,5	0,0			CONTENTS OF INDEX REGISTER
0,0,6,6	2,7	B.E.O		GO TO DONE (BRANCH FORWARD)
0,0,6,7	0,4			
0,0,6,8	1,B	A.B.A		ADD B REG. TO A - RESULTS IN A
0,0,6,9	0,8	I.N.X		INCREMENT THE INDEX REGISTER
0,0,6,A	2,0	B.R.A		GO TO ADD (BRANCH BACK TO 0064
0,0,6,B	F,8			(6 LOCATIONS + 2 = 08 so F8
0,0,6,C	9,7	S.T.A.A		DONE
0,0,6,D	8,0			
0,0,6,E	3,F	S.W.I		

(You should check these relative offsets — 04 and F8)

Note: ABA adds the A and B accumulators and places the result in A. (This is the only instruction that will do this.)

2. Rerun the program as before except put 0 in the location after the last item. What happens if you forget the final 0? What happens if you place 0 in memory location 82?

E. Forming a Checksum (a "logical" sum)

Return to the last program and change the program so that it EORs (Exclusive OR) the numbers together instead of adding them. (This instruction will ~~EOR~~ the contents of the A and B registers.) This result is called a logical sum or checksum, and is often used to check for errors in tape records. (See page 108 of BASIC MICROPROCESSORS and THE 6800.)

Note: EOR is the same as addition except that there are no carries.

(Truth Table)

A	B	SUM	CARRY	$A \oplus B$
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	0	1	0

F. Displaying an Array

1. Another example of indexing may be used to place different data on each of the six LED displays. The following program will help demonstrate:

Note: This is a very difficult example program.

Location	Hex	Instr.	Operand # , X	Comment
0,0,6,0	C,6	L,D,A	B # 0,2	TURN ON LED-1 START(LOAD B
0,0,6,1	0,2			WITH 02)
0,0,6,2	F,7	S,T,A	B	STORE B in E400 REGISTER
0,0,6,3	E,4			
0,0,6,4	0,0			
0,0,6,5	C,E	L,D,X		START OF ARRAY
0,0,6,6	0,0			
0,0,6,7	8,0			
0,0,6,8	A,6	L,D,A	A 0,0,X	GET DATA DISPLAY
0,0,6,9	0,0			

(continued on next page)

0 0,6 A	B,7	S T A A		SEND DATA TO DISPLAY
0 0,6 B	E,4			
0 0,6 C	0,2			
0 0,6 D	D,F	S T X	9,0	SAVE X
0 0,6 E	9,0			
0 0,6 F	C,E	L,D X		SET DELAY COUNTER
0 0,7 0	0,0			
0 0,7 1	0,0			
0 0,7 2	0,8	I,N X		DELAY
0 0,7 3	2,6	B,N E		NOT ZERO, GO TO DELAY
0 0,7 4	F,D			
0 0,7 5	D,E	L,D X	9,0	
0 0,7 6	9,0			
0 0,7 7	0,8	I,N X		NEXT DATA IN ARRAY
0 0,7 8	7,8	A,S L		TURN ON NEXT DISPLAY
0 0,7 9	E,4			
0 0,7 A	0,0			
0 0,7 B	2,A	B,P L		IF DONE, GO TO DISPLAY
0 0,7 C	E,B			
0 0,7 D	7,E	J,M P		JUMP TO START
0 0,7 E	0,0			
0 0,7 F	6,0			

(Look up any instructions you are not sure of.)

The program starts with a 1 in bit-1 of location E400 and continues until that 1 moves into the negative (or sign bit). ASL shifts the contents of E400 left one bit and clears the empty bit. BPL branches back if bit-7 is still 0. Remember that the System X does not use bits-0 and 7 of memory location E400

2. Run the program with the following data:

Display 1 = (80) = 6E

Display 2 = (81) = 9E

Display 3 = (82) = 1C

Display 4 = (83) = 1C

Display 5 = (84) = FC

Display 6 = (85) = 00

Notice what happens when you reduce the delay by changing memory location 0070 to 80, E0, F0, FC, FF? Try some of your own.

G. Forming Arrays Using the System X

1. Now that you have seen how you can handle an array, the next question is how to form one. This procedure will require a counter and a pointer. The pointer contains the address of the next empty location in the array; the counter contains the length of the array. (An example to guide you through each step will help.)
2. The basic Procedure is:

First Step

Initialization

Pointer = Start of Array (Set of Data)
Counter = 0

Second Step

Place Data in Array

(Pointer) = Data. Remember that the parentheses mean "contents of."

Third Step

Update Counter and Pointer

Pointer = Pointer + 1
Counter = Counter + 1 (Increment)

The above procedure assumes that you have all the data available. It also provides no way to end the array formation.

One method is:

Fourth Step Check to See if Enough Data has been Collected

If Counter = length, then done; otherwise, return to Second Step.

The array is formed. The next step is to clear the array.

H. Clearing an Array (or set of data from memory)

1. The following example clears an area of memory; in this case memory locations 80 through 87. Note the appearance of the NOP (No Operation). It does nothing except to make the program easier to change. (It is inserted as a convenience.)

Note: Look up all the instructions in the 6800 manuals.

Location	Hex	Instr.	Operand # , X	Comment
0 0, 6 0	4 F	C L R A		
0 0, 6 1	0 1	N O P		
0 0, 6 2	C E	L D X	# S 8 0	POINTER START OF ARRAY
0 0, 6 3	0 0			
0 0, 6 4	8 0			
0 0, 6 5	C 6	L D A B	= 8	LENGTH OF ARRAY IS 8
0 0, 6 6	0 8			
0 0, 6 7	A 7	S T A A	0 0, X	CLEAR AN ELEMENT CLEAR
0 0, 6 8	0 0			
0 0, 6 9	0 8	I N X		
0 0, 6 A	5 A			DECREMENT B
0 0, 6 B	2 6	B N E		NOT EQUAL, GO TO CLEAR
0 0, 6 C	F A			
0 0, 6 D	3 F			

2. Now enter the program and run it. Now try a few changes to have the program do the following tasks, and then test each revision:
 - a. Clear memory locations 80 through 89.
 - b. Place 80 HEX in memory locations 90 through 99.

I. Placing Usable Values in an Array (This is the Fun Part)

1. A natural continuation is to put actual numbers in each entry. Modify the original program by changing STAA X to STAB X (i.e., E7 instead of A7)? How would you change the program to reverse the order of the numbers? **Hint:** Use INCA but remember to adjust the offset in BNE CLEAR.
2. Another examine program will set each element to twice the preceding element, starting with 1. Do you see any new instruction? How about 5A?

Location	Hex	Instr.	Operand # , X	Comment
0 0 6 0	8 6	L D A A	# 1	FIRST ELEMENT IS 1
0 0 6 1	0 1			
0 0 6 2	C E	L D X	# \$ 0	POINTER
0 0 6 3	0 0			
0 0 6 4	8 0			
0 0 6 5	C 6	L D A B	# 8	LENGTH OF ARRAY IS 8
0 0 6 6	0 8			
0 0 6 7	A 7	S T A A	0 0 X	PLACE ELEMENT IN ARRAY SETUP
0 0 6 8	0 0			
0 0 6 9	4 8	A S L A		NEXT ELEMENT IS ELEMENT X2
0 0 6 A	0 8	I N X		
0 0 6 B	5 A	D E C B		
0 0 6 C	2 6	B N E		GO TO SETUP
0 0 6 D	F 9			
0 0 6 E	3 F			

3. This program may be written so as to eliminate a counter. This can be done by just letting the 1-bit, in position 0 move across the word. Try to modify the program to eliminate the counter. The program format may be shown as follows:

(80) = 80 (10000000)
(81) = C0 (11000000)
(82) = E0 (11100000)
(83) = F0 (11110000)
(84) = F8 (11111000)
(85) = FC (11111100)
(86) = FE (11111110)
(87) = FF (11111111)

Hint: Use Arithmetic Shift Right? Can you see why this instruction is part of the set?

J. Entering Input Data Into an Array

1. The next step is to form an array from the keyboard using input data. To simplify the program, just use keys "0" — "7".
2. The routine for forming the array will be:

First Step	Initialization
Pointer	= Start of Array (40)
Counter	= Length of Array (4)
Second Step	Wait for Key to be Pressed
Third Step	Debounce Key with 1 ms. Delay
Fourth Step	Identify Key
Fifth Step	Place Key in Array
(Pointer)	= Key
Sixth Step	= Update Counter and Pointer
Pointer	= Pointer + 1
Counter	= Counter + 1
Seventh Step	Wait for Key to be Released
Eighth Step	If Counter $\neq$ 0, Return to Step 2

(See the flowchart for forming an array, Figure 8 - 2.)

(This is a very long program, and the flowchart is necessary)

Location	Hex	Instr.	Operand # , X	Comment
0 0 6 0	C E	L D X	# A 0	POINTER
0 0 6 1	0 0			
0 0 6 2	A 0			
0 0 6 3	D F	S T X		
0 0 6 4	B 0			
0 0 6 5	8 6	L D A A	# 0 4	COUNTER
0 0 6 6	0 4			
0 0 6 7	9 7	S T A A		
0 0 6 8	B 2			
0 0 6 9	B 6	L D A A		GET KEYS "0" to "7" WAIT
0 0 6 A	E 0			
0 0 6 B	0 0			
0 0 6 C	8 1	C M P A	# \$ F F	ARE ANY KEYS PRESSED
0 0 6 D	F F			
0 0 6 E	2 7	B E Q		IF NO KEYS PRESSED, GO TO WAIT
0 0 6 F	F 9			
0 0 7 0	C E	L D X	# \$ 7 C	1 ms. FOR DEBOUNCE DELAY
0 0 7 1	0 0			
0 0 7 2	7 C			
0 0 7 3	0 9	D E X		
0 0 7 4	2 6	B N E		GO TO DELAY
0 0 7 5	F D			
0 0 7 6	5 F	C L R B		KEY NUMBER = 0
0 0 7 7	4 4	L S R A		IS NEXT BIT = 0 SR KEY

(continued on next page)

SA DEC B

so NEG B

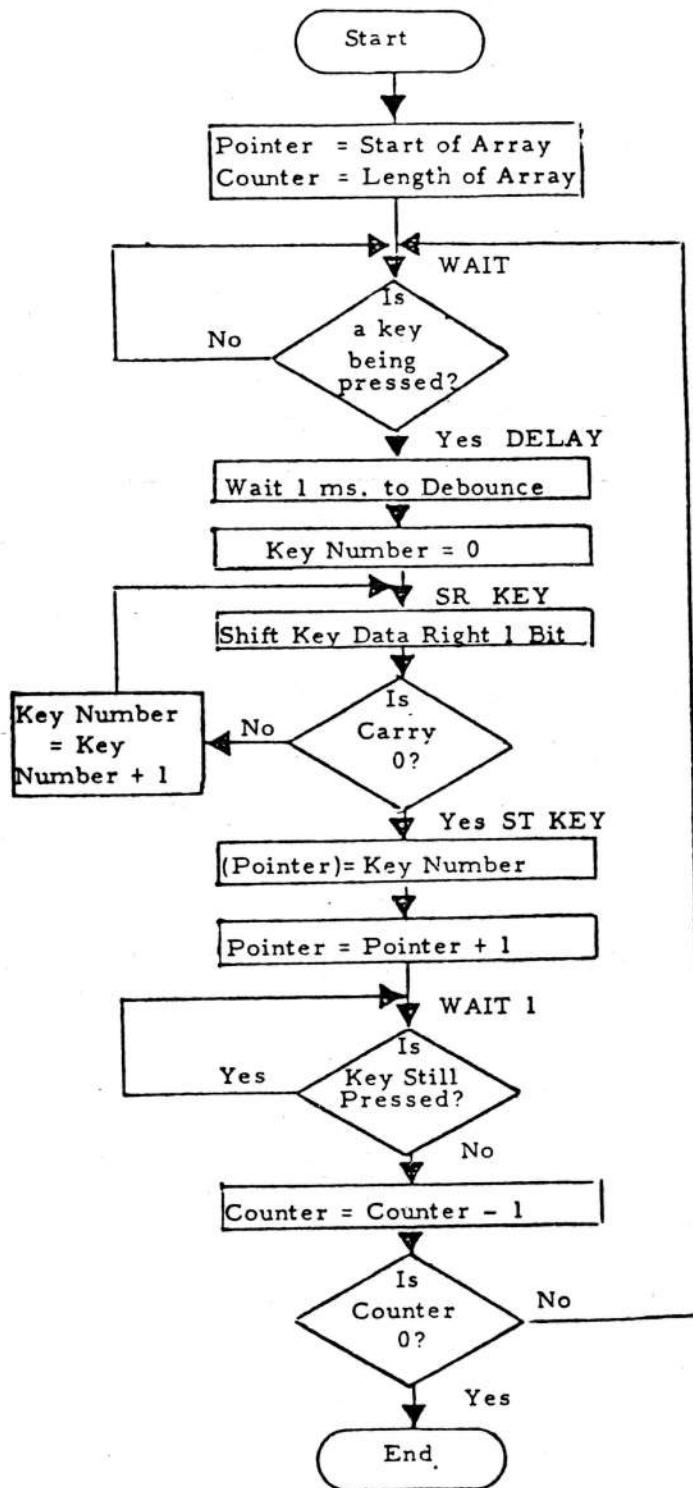
D8

0 0, 7 ⁸	2 4				YES, DONE
0 0, 7 ⁹	0 3				
0 0, 7 ^A	5 C I N C B				NO KEY NUMBER = KN+1
0 0, 7 ^B	2 D B R A				GO TO SR KEY
0 0, 7 ^C	F A				
0 0, 7 ^D	D E L D X		\$ B 0		LOAD X WITH POINTER
0 0, 7 ^E	B 0				
0 0, 7 ^F	E 7 S T A B		0 0, X		
0 0, 8 ⁰	0 0				
0 0, 8 ¹	0 8 I N X				UPDATE AND SAVE POINTER
0 0, 8 ²	D F S T X		\$ B 0		
0 0, 8 ³	B 0				
0 0, 8 ⁴	B 6 L D A A				GET KEYS "0" to "7" WAIT 1
0 0, 8 ⁵	E 0				
0 0, 8 ⁶	0 0				
0 0, 8 ⁷	8 1 C M P A # \$ F F				ARE ANY KEYS PRESSED
0 0, 8 ⁸	F F				
0 0, 8 ⁹	2 6 B N E				YES, WAIT UNTIL ALL KEYS RELEASED
0 0, 8 ^A	F 9				
0 0, 8 ^B	7 A D E C				DECREMENT COUNTER
0 0, 8 ^C	0 0				
0 0, 8 ^D	B 2				
0 0, 8 ^E	2 6 B N E				GO TO WAIT
0 0, 8 ^F	D 9				
0 0, 9 ⁰	3 F S W I				

(Don't forget to look up all new instructions.)

3. Enter the program and run it. Try several different key sequences. Remember, the pointer starts at \$00A0 memory location. Examine that location.
4. Save "6" key number in memory location \$A0 through \$A5.
5. Save "4" key number in memory location \$A8 through \$AB.
6. What happens when you press several keys at once?

Flowchart for Forming Array from Keyboard



Chapter 9: Advanced Input/Output Formating

Objective: The student will become familiar with key identification, debouncing, shift instructions, keyboard response, and strobe flags.

Student study requirements: Pages A37 and A38 of the M6800 PROGRAMMING MANUAL.

Discussion

I. General Information on Input/Output

Input and output as discussed in Chapter 7, in general terms, are those devices through which the MPU has access to the outside world. Input and output are similar to memory accesses. The processor can transfer data to and from peripherals in the same way it can transfer data to and from memory. In fact memory, the keyboard, the displays, PIA, and ACIAs are all simple peripherals. For this reason, we feel that the more proficient you become in handling data to and from the keyboard and displays, the more proficient you will be in handling data outside the System X microcomputer.

Peripherals vary tremendously. They may be electronic, mechanical, or electro-mechanical. They may operate on a digital voltage or an analog signal. A simple I/O section might include a pressure sensor that provides data every five minutes; a printer that transfers 100 bits per second; or a disk that transfers 250,000 bits per second. The data on input lines will, of course, change independently of the computer. Signals may be necessary to hold or transform data and control the operating modes of peripherals.

Few standards exist; each peripheral is a unique problem. A special interface must translate between the signals the processor uses and those the peripheral uses. The interface must provide proper timing and control. With the availability of single chip computers and large memories, the input/output section is the most expensive part of many microcomputers.

Since input/output is such a complex subject, we have chosen to handle the matter in four different chapters. Also, all laboratory exercises are to develop your skill for the handling of data to and from the microcomputer.

II. New Instructions

TST (Test): This sets the sign and zero flags depending on the contents of the specified accumulator on memory address, clears the overflow and carry flags. (See page A73 of the M6800 PROGRAMMING MANUAL.)

Example: Suppose accumulator B contains \$42, after executing TSTB the sign, zero, overflow and carry status are 0.

0100 0010 Non-zero value
Z set to 0

Set S to 0

III. Laboratory Exercise

A. Using the System X Keyboard

1. It should be reinforced at this time that the reason so much time is spent studying the manipulation of the keyboard and display is they are the outside world to the MPU, and the keyboard is used for both control and data entry. Therefore, the use of the keyboard does require programming, and since we are working with binary information at the same time, we study bit-by-bit control.

Table 9 - 1
Patterns Produced by Various Key Closures

Key Pressed	Pattern	
	Binary	Hex
0	11111110	FE
1	11111101	FD
2	11111011	FB
3	11110111	F7
4	11101111	EF
5	11011111	DF
6	10111111	BF
7	01111111	7F
None	11111111	FF

2. Table 9 - 1 contains the binary patterns which are produced in memory location E000 by pressing the various keys "0" to "7". If no keys are pressed, the pattern is all 1's (i.e. FF hexadecimal). So the following program will wait until you press a key in the "0" to "7" group.

Location	Hex	Instr.	Operand # , X	Comment
0 0 , 6 0	B 6	L D A	A E 0 0 0	GET KEYS "0" - "7" WAIT
0 0 , 6 1	E 0			
0 0 , 6 2	0 0			
0 0 , 6 3	8 1	C M P A	# \$ F F	ARE ANY KEYS PRESSED
0 0 , 6 4	F F			
0 0 , 6 5	2 7	B E Q		NO, GO TO WAIT
0 0 , 6 6	F 9			
0 0 , 6 7	3 F	S W I		

Note: By use of the single step instruction, you should be capable of determining just what happens with the instruction CMPA#\$FF. Also note that the value is not stored in any location, so to find the value of the key depressed, you will have to examine the A accumulator.

3. Enter and run the program. Now change the program so that it waits until you stop pressing a key.
4. Test the new program and see what happens if you press a new key before releasing the old one.
5. What happens if you press several keys at once?
6. Now combine the two programs with the original version first. The combined program should wait for you to press a key and then release it.

Location.	Hex	Instr.	Operand # , X	Comment
0 0 6 0	B 6	L D A A	E 0 0 0	GET KEYS "0" - "7" WAIT
0 0 6 1	E 0			
0 0 6 2	0 0			
0 0 6 3	4 3	C O M		COMPLEMENT KEYS
0 0 6 4	8 8	E O R A		EXCLUSIVE OR KEYS
0 0 6 5	0 0			
0 0 6 6	2 7	B E Q		NO, GO TO WAIT
0 0 6 7	F 8			
0 0 6 8	B 6	L D A A	E 0 0 0	WAIT 1
0 0 6 9	E 0			
0 0 6 A	0 0			
0 0 6 B	8 1	C M P A	# \$ F F	YES, WAIT FOR IT TO BE RELEASED
0 0 6 C	F F			
0 0 6 D	2 6	B N E		GO TO WAIT 1
0 0 6 E	F 9			
0 0 6 F	3 F			

7. You will notice in this program we have used the COM and EOR instructions. Try setting a break point and see the effect of these instructions on the A accumulator and the CCR.

B. Debouncing Keys and Timing Delays

1. As you have tried the previous programs you have found sometimes they work, and other times you have funny things happen. An example would be to depress the "7" key and after the System X completes the SWI, the display will show — — — PC. In other words, the System X cannot tell what you want since it is seeing the effects of key bounce. Even though the System X has the best mechanical keyboard available today, it still has the problem of mechanical bounce. If you will look on page 005 of the ASCII T68 bug monitor printout, you will notice on line 0211A or memory location \$ F8E3 a delay program for the purpose of key debouncing.

2. Let's examine this routine and see how we can effectively determine the amount of time we would be in a delay loop.

<i>F8E3</i>	CE 0000	LDX # 0
<i>F8E6</i>	09	DEX
<i>F8E7</i>	26 Fd	BNE to <i>F8E3</i>

The instruction LDX # 0 will load the X register with all \$0000. The instruction DEX will decrement the X register to FFFF, and the instruction BNE will branch back to DEX until the X register becomes all zeros again. If you now look-up these instructions on the repertoire card, you will find they require the following number of clock cycles. Look-up the cycle column (~) for the number of cycles, i.e. LDX requires 3 cycles, DEX requires 4 cycles, and BNE requires 4 cycles. Since the System X operates on a 1 Mhz clock or 1 microsecond period, the time delay could be calculated by the following:

time delay = 1 microsecond (\$FFFF(4 + 4) + 3)

convert \$FFFF to decimal = 65535

time delay = 1 microsecond (65535 • 8) + 3

time delay = 524283 microseconds or .524 seconds

3. Since a delay of 1/2 second is quite long for key debounce, let's calculate a time delay of 1 millisecond or 1000 microseconds stated as follows:

1000 microseconds = 1 microsecond (X(4 + 4) + 3)

or 8X + 3 = 1000 microseconds

8X = 997 microseconds

X = 124

Convert 124 to HEX = \$7C

so a 1 millisecond delay would be

LDX # \$7C

DEX

D LOOP

BNE

D LOOP

4. Code this program and add the delay routine between the two sections of the previous program. The System X should now wait for you to press and release a key.
5. Try rewriting the program for longer and shorter time delays.
6. Rewrite the programs so as to store the binary information received from the key closure.

C. Key Identification

1. The next question is how to transform a key closure into the corresponding digit. Look at Table 9 - 1 again. The bit which is 0 is the one which identifies the key, i.e. bit-0 is 0 for key "0", bit-1 for key "1", etc. So all you have to do is figure out which bit is 0. You can do that by counting the number of shifts required to get a 0-bit into the carry, (i.e. if accumulator A contains the key closures from memory location E000:

	CLRB	KEY NUMBER = 0
SRKEY	LSRA	IS NEXT BIT 0?
	BCC DONE	YES, DONE
	INCB	NO, KEY NUMBER=KEY NUMBER+1
	BRA SRKEY	
DONE	SWI	

Accumulator B contains the key number at the end of the program.
(See the flowchart in Figure 9 - 1.)

2. An alternative identification program uses somewhat different initial conditions in order to eliminate one of the jump instructions:

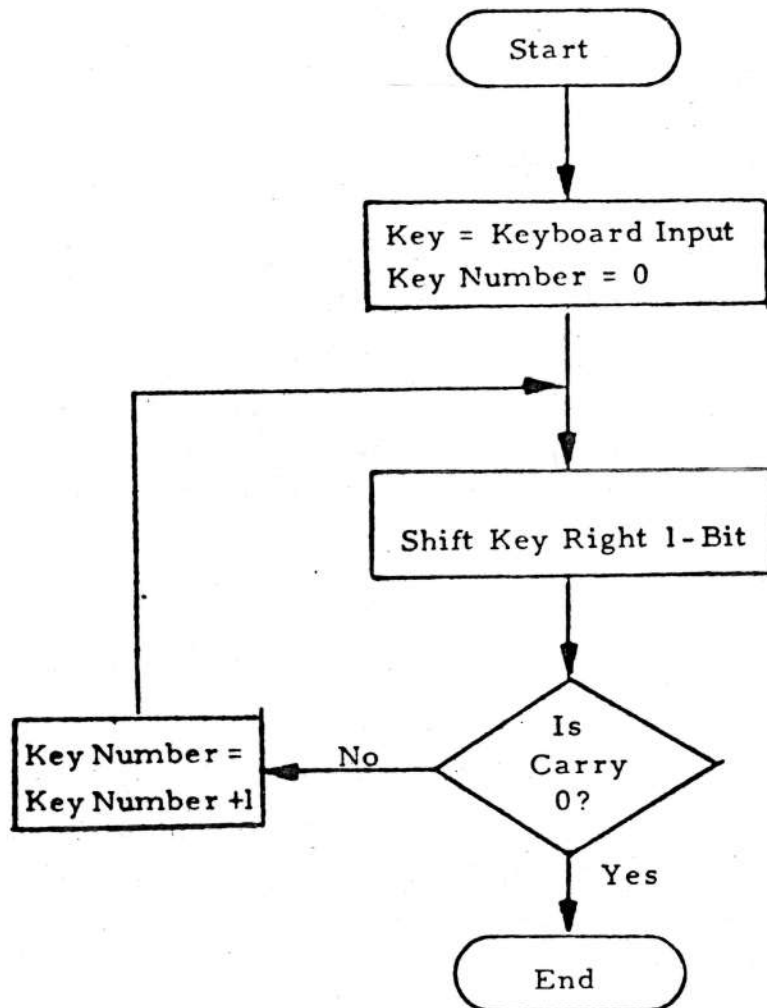
	LDAB #\$FF	KEY NUMBER = -1
SRKEY	INCB	KEY NUMBER=KEY NUMBER+1
	LSRA	IS NEXT BIT 0?
	BCS SRKEY	NO, KEEP LOOKING FOR 0—BIT
	SWI	

The entire program is:

- a. Wait for a key closure
- b. Wait 1 ms. to debounce the key
- c. Identify key

Location	Hex	Instr.		Operand # , X	Comment
0 0 6 0	B 6	L D A	A	E 0 0 0	GET KEYS "0" - "7" WAIT
0 0 6 1	E 0				
0 0 6 2	0 0				
0 0 6 3	8 1	C M P	A	# \$ F F	ARE ANY KEYS PRESSED
0 0 6 4	F F				
0 0 6 5	2 7	B E Q			NO GO TO WAIT
0 0 6 6	F 9				
0 0 6 7	C E	L D X		# \$ 7 C	
0 0 6 8	0 0				

Figure 9 - 1
Flowchart for Key Identification



0 0,6 9	7 C				
0 0,6 A	0 9				DELAY
0 0,6 B	2 6				
0 0,6 C	F D				
0 0,6 D	5 F	C L R B			KEY NUMBER IS ZERO
0 0,6 E	4 4	L S R A			IS NEXT BIT ZERO SR KEY
0 0,6 F	2 4	B C C			YES, GO TO DONE
0 0,7 0	0 3				
0 0,7 1	5 C	I N C R		NO	KEY NUMBER IS KEY NUMBER + 1
0 0,7 2	2 0	B R A			GO TO SR KEY
0 0,7 3	F A				
0 0,7 4	D 7	S T A B	\$ 8 0		DONE
0 0,7 5	8 0				
0 0,7 6	3 F	S W I			

(This program saves the key code at memory location \$0080.)

3. The following program will respond to all 16 keys. Draw a flowchart and write an assembler language program for this machine language program.

Location	Hex	Instr.	Operand # , X	Comment
0 0,6 0	b 6	L D A	A E 0 C 0	GET 0 - 7 KEYS WAIT 0
0 0,6 1	E 0			
0 0,6 2	0 0			
0 0,6 3	8 1	C M P	# F F	
0 0,6 4	F F			
0 0,6 5	2 6	B N F		BRANCH TO DLY 2
0 0,6 6	0 d			
0 0,6 7	C E	L D X	0 0,7 d	LOAD INDEX DLY 1
0 0,6 8	0 0			

0 0,6 ⁹	7.d				
0 0,6 ^A	0.9	D.E.X			DECREMENT INDEX REGISTER
0 0,6 ^B	2.6	B.N.E			
0 0,6 ^C	F.d				
0 0,6 ^D	b.6	L.D.A	A	E.0.0.2	GET 8-F KEYS WAIT 1
0 0,6 ^E	E.0				
0 0,6 ^F	0.2				
0 0,7 ⁰	8.1	C.M.P		F.F	
0 0,7 ¹	F.F				
0 0,7 ²	2.7	B.E.0			BRANCH TO WAIT 0
0 0,7 ³	E.C				
0 0,7 ⁴	C.E	L.D.X		0 0,7.d	LOAD INDEX DLY 2
0 0,7 ⁵	0.0				
0 0,7 ⁶	7.d				
0 0,7 ⁷	0.9	D.E.X			DECREMENT INDEX REGISTER
0 0,7 ⁸	2.6	B.N.E			
0 0,7 ⁹	F.d				
0 0,7 ^A	b.6	L.D.A	A	E.0.0.C	ARE 0 - 7 RELEASED WAIT 2
0 0,7 ^B	E.0				
0 0,7 ^C	0.0				
0 0,7 ^D	8.1	C.M.P		F.F	
0 0,7 ^E	F.F				
0 0,7 ^F	2.6	B.N.E			BRANCH TO DLY 2
0 0,8 ⁰	F.3				
0 0,8 ¹	b.6	L.D.A	A	E.0.0.2	ARE 8 - F RELEASED WAIT 3
0 0,8 ²	E.0				
0 0,8 ³	0.2				
0 0,8 ⁴	8.1	C.M.P	#	F.F	
0 0,8 ⁵	F.F				
0 0,8 ⁶	2.6	B.N.E			BRANCH TO DLY 2
0 0,8 ⁷	E.C				
0 0,8 ⁸	3.F				SWI

D. System X Displays from Program

1. We saw earlier how to turn the displays on and off, and how to pulse and scan them. Now we will discuss how to send data to the displays, and how to combine display control with other tasks.
2. The following program places the contents of memory location 80 on all the displays for one second:

Location	Hex	Instr.	Operand # , X	Comment
0 0 6 0	8 6	L D A	A # . \$. F F .	TURN ON LED
0 0 6 1	F F			
0 0 6 2	B 7	S T A	A	
0 0 6 3	E 4			
0 0 6 4	0 0			
0 0 6 5	D 6	L D A	B . \$. 8 . 0 .	GET DATA
0 0 6 6	8 0			
0 0 6 7	F 7	S T A	B	STOR TO LEDS
0 0 6 8	E 4			
0 0 6 9	0 2			
0 0 6 A	C E	L D X	# . \$. F 4 2 3	DELAY $\frac{1}{2}$ SECOND DELAY 1
0 0 6 B	F 4			
0 0 6 C	2 3			
0 0 6 D	0 9	D E X		DECREMENT 1
0 0 6 E	2 6	B N E		NO. GO TO DECREMENT 1
0 0 6 F	F D			
0 0 7 0	C E	L D X	# . \$. F 4 2 3	DELAY 2
0 0 7 1	F 4			
0 0 7 2	2 3			
0 0 7 3	0 9	D E X		DECREMENT 2
0 0 7 4	2 6	B N E		NO EQUAL, GO TO DECREMENT 2
0 0 7 5	F D			
0 0 7 6	3 F			

a. The program calls for a one second delay in the readout. Since the System X has a 1Mhz clock, this would call for the value \$1E846 to be loaded into the index register. The largest value that may be loaded in any register is \$FFFF, so we must divide the delay into two different delays of \$F423 each (1/2 second). In later chapters, you will find how to write a subroutine to accomplish the same thing.

b. Enter the run this program. Try the following data in memory location \$80: FC, 60, dA, F2, 66, b6, bE, EO, F3, F6. Make up some displays of your own.

Tables 9 - 2 and 9 - 3 contain the codes for hexadecimal digits, some letters, and symbols.

Table 9 - 2
Seven-Segment Code Table
Hexadecimal Digits

Digits	Hex Code
0	FC
1	60
2	dA
3	F2
4	66
5	b6
6	bE
7	EO
8	FE
9	E6
A	EE
b	3E
C	9C
d	7A
E	9E
F	8E

Table 9 - 3

Seven-Segment Code Table
Other Symbols

Capital Letters

Letter	Hex Code
A	EE
C	9C
E	9E
F	8E
H	6E
J	78
L	1C
O	FC
P	CE
U	7C
Y	76

Lower case letters

Letter	Hex Code
b	3E
c	1A
d	7A
h	2E
n	2A
o	3A
r	0A
u	38

Symbols

Symbol	Hex Code
	CA
	02
	10

3. You can solve the conversion problem by just placing Table 9 - 2 in memory. Put it in memory location 0084 through 0093. The following program will convert a hexadecimal digit in memory location \$90 to a seven-segment code in memory location \$91.

Location	Hex	Instr.	Operand # , X	Comment
0,0,6 0	C E	L D X	# \$ 9 4	
0,0,6 1	0 0			
0,0,6 2	9 4			
0,0,6 3	D F	S T X		SAVE BASE ADDRESS OF TABLE
0,0,6 4	9 2			
0,0,6 5	9 6	L D A A		GET HEXADECIMAL DIGITS
0,0,6 6	9 0			
0,0,6 7	9 B	A D D A		USE DIGIT TO INDEX TABLE
0,0,6 8	9 3			
0,0,6 9	9 7	S T A A		
0,0,6 A	9 3			
0,0,6 B	D E	L D X		GET INDEXED ADDRESSES
0,0,6 C	9 2			
0,0,6 D	A 6	L D A A	0 0 X	AND USE IT TO FETCH 7-SEG. CODE
0,0,6 E	0 0			
0,0,6 F	9 7	S T A A		
0,0,7 0	9 1			
0,0,7 1	3 F	S W I		

4. Enter the program and run it for all hexadecimal digits. Normally the tables of codes will be in ROM or PROM since it never changes. In fact, the ASCII T68 bug has the table starting memory location \$FF51. (See the USER'S MANUAL printout of the ASCII T68 bug.)
5. Revise the program so it uses the ASCII T68 bug.

E. Counting On the Displays

- Now combine the display program and the table so the program counts on the displays. Enter and run the program.

Location	Hex	Instr.	Operand # , X	Comment
0 0 7 0	8 6	L D A A	# S F F	
0 0 7 1	F F			
0 0 7 2	B 7	S T A A	E 4 0 0	TURN ON LEDS
0 0 7 3	E 4			
0 0 7 4	0 0			
0 0 7 5	C E	L D X	F F 5 1	GET BASE ADDRESS of TBLS COUNT
0 0 7 6	F F			
0 0 7 7	5 1			
0 0 7 8	A 6	L D A A	0 0 X	GET DATA NEX DISPLAY
0 0 7 9	0 0			
0 0 7 A	B 7	S T A A	E 4 0 2	
0 0 7 B	E 4			
0 0 7 C	0 2			
0 0 7 D	D F	S T X		STORE DATA
0 0 7 E	A 0			
0 0 7 F	C E	L D X		WAIT 1/2 SECOND DELAY 1
0 0 8 0	F 4			
0 0 8 1	2 3			
0 0 8 2	0 9	D E X		
0 0 8 3	2 6	B N E		
0 0 8 4	F D			
0 0 8 5	C E	L D X		DELAY 2
0 0 8 6	F 4			
0 0 8 7	2 3			
0 0 8 8	0 9	D E X		
0 0 8 9	2 6	B N E		
0 0 8 A	F D			
0 0 8 B	D E	L D X		FETCH DATA
0 0 8 C	A 0			
0 0 8 D	0 8	I N X		
0 0 8 E	8 1	C M P A	# S 8 E	WAS COUNT F
0 0 8 F	8 E			
0 0 9 0	2 6	B N E		NO, GO TO NIX DISPLAY
0 0 9 1	E 6			
0 0 9 2	2 0	B R A		YES, GO TO COUNT. START
0 0 9 3	E 1			OVER AT ZERO

- Change the counting delay and see what happens? Note that we started at \$0070 which will allow for additions between \$0060 and \$006F

3. Change the program so it counts down instead of up. Remember to change the starting address to the end of the table (\$FF60) the direction of updating (DEX instead in INX), and the final comparison.
4. Now change the program so it waits for key "F" to be pressed, then counts up continuously. Looking at key "F" is simple, i.e.,

```

006B B6 LDAA IS "F" PRESSED? WAIT
006C E0
006D 02
006E 2B BMI NO, WAIT
006F FB

```

5. Change the program so that it waits for key "B" to be pressed, and then counts down continuously. Checking for key "B" requires a logical AND.

```

WAITB LDAA $8006 GET KEYS "8—F"
      ANDA #00001000 IS KEY "B" PRESSED?
      BNE WAITB NO, WAIT

```

6. Change the program so that it counts up but checks key "B" after each one second delay, and waits as long as key "B" is pressed. Remember to use accumulator B since A holds the seven-segment code. You can improve the responsiveness of the system by dividing the delay into tenths of a second. The following program is a tenth of a second delay:

```

      LDX #30D2
DLYT DEX
      BNE DLYT

```

Introduce the tenth of a second response. (Use A as a counter, but save its old contents!) Can you notice the difference? Clearly a computer does not have to check keys very often to provide almost instantaneous response as far as an operator is concerned.

7. Finally, change the program so that it checks keys "B" and "F" every second. Key "B" causes the program to wait, while key "F" causes it to resume counting. How would you modify the program so that key "F" acts as a stop/start button, (i.e. the first closure stops the program, the second closure restarts it, etc.)?

Chapter 10: Programming Techniques

Objective: To familiarize the student with stages of program development, flowcharting, System X save stack, System X return from interrupt, program debugging, software support tools, binary arithmetic, decimal arithmetic, and multi-word arithmetic.

Student study requirements: Pages A4, A27, A54-56, A69, and A71 of the M6800 PROGRAMMING MANUAL; Chapter 6 of BASIC MICRO-PROCESSORS and THE 6800 by Bishop; review Chapter 6 of this manual.

Discussion

I. Analyzing Software Problems (debugging)

Now that you have seen and written some fairly complex programs, you are probably wondering how to develop a general procedure to design software to solve a problem. This procedure is completely machine independent, and can be applied to any software problems you are likely to encounter. The most important thing to remember about this procedure is that you **do not** concern yourself with the programming language details until well into the solution. This is true of even the seemingly "trivial" programs. There is no way more certain to result in a program that is sloppy, ill designed, and hard to debug than to try to write the program directly from the problem definition. To be effective, software must be designed first and then implemented using correct techniques.

The systematic approach to developing a programmed system is a logical extension of the normal problem solving cycle engineers and scientists have employed for years. It consists of seven basic steps:

- problem definition
- problem partitioning
- algorithm development for each partition (program design)
- writing the program for each partition (coding)
- debugging each program
- integrating the programs back into the system
- final system debug

Using this technique, the problem is broken into smaller and smaller sub-problems until they are a size which you can deal with conveniently and effectively. This is because it is much easier to focus your attention on one small section of the system at a time. You develop each of these blocks and sub-blocks into a group of detailed flowcharts and programs, each of which is tested and debugged. They are then interfaced and the whole system tested. This systematic approach is intended to help you minimize errors, since the small highly localized programs are much easier to thoroughly check out than a single, large, spread-out program.

You start with a central problem and partition it into logical blocks, solve and debug each of the blocks, and finally integrate and refine the blocks into the final system. There may be one or many levels of blocking, depending on the complexity of the problem. With experience, you will find this general approach to be the most direct and consistent way to implement a working software system, regardless of size. Less organized approaches may work for smaller systems, but will hopelessly tangle as the systems grow in size. It is best to learn the general procedure and use it on all problems, small or large. The greatest disasters usually occur when the whole design procedure is dispensed with because the problem is too "trivial" to warrant the general approach. Conversely, dogged application of this approach can make many formidable problems turn out far better than anticipated.

The fact that most microcomputer programmers are not professional programmers is fairly obvious. Most current microcomputer programmers are logic designers, engineers, or technicians, and many are programming for the first time. Since they probably will be forced to use machine and assembler languages, these users will be learning programming, algorithm development, and machine structure, all at the same time. The use of assembler language programming and flowcharts will enable you to separate these learning activities. In particular, the initial process of learning general algorithm development is better presented with general flowcharts than one specific language.

Once the problem is broken into blocks; algorithm developed, flowcharts completed, defined in assembler, and written in machine language; it must then be tested and debugged. We will help you develop these skills in this chapter.

II. New Instructions

- A. CLC (Clear Carry):** This instruction sets the carry bit of the Conditional Code Register (CCR) to zero. No other status registers or contents are affected. (See Page A27 of the M6800 PROGRAMMING MANUAL.)
- B. ADC (Add with Carry):** This instruction adds the contents of the Carry bit (C) from the conditional code register to the contents of the selected accumulator, plus the contents of selected memory. Example: Suppose the A accumulator contains \$3C, memory location \$0004 contains \$7A, and the carry bit is 1 set with the following program:

MEMORY ADDRESS	MEMORY CONTENTS
0000	\$89 ADCA
0001	\$04

The contents of 04 or \$7A would be added to the A accumulator or \$3C.

$$\begin{array}{rcl} \$7A = & \% & 01111010 \\ \$3C = & \% & 00111100 \\ \text{Add} & & 10110110 \\ \text{Add Carry} & & \underline{1} \\ & \% & 10110111 = \$b7 \end{array}$$

So the reset \$b7 would be found in the A accumulator.

1. The ADC Command uses four methods of addressing memory, which are:
 - Direct
 - Immediate
 - Extended
 - Indexed

This instruction will be used most often with multi-byte additions to include the carry in addition of second and subsequent bytes.

2. Since the carry flag initially can be either set or clear, the programmer must control its condition. The CLC instruction should be used prior to an ADC instruction.

- C.** ROL (Rotate Left): This instruction will shift all bits of the selected accumulator or selected memory one place to the left, with bit-0 loaded from the carry bit of the CCR, and bit-7 of the original word shifted into the carry bit of the CCR. (See page A54 of the M6800 PROGRAMMING MANUAL.) Call the students' attention to the drawing on the instruction card under "Boolean Operation" opposite ROL.

Example:

Suppose the A accumulator contains the value \$7C, and the C-bit is set to 1 after instruction ROLA:

$$\begin{array}{rcl} \text{A original } \$7C = & \% & 01111100 \\ \text{After ROLA} & = & \% 11111001 \quad \text{Carry Bit} \end{array}$$

After ROLA, A = \$F9 and the C-bit of the CCR will equal 0 since bit-7 was shifted into the C-bit.

- D. ROR (Rotate Right):** This instruction will shift all bits of the selected accumulator or selected memory one place to the right, with bit-7 being loaded from the carry bit of the CCR, and bit-0 being loaded into the carry bit of the CCR. (See page A55 of the M6800 PROGRAMMING MANUAL.)

Example:

Suppose the A accumulator contains the value \$7C, and the C-bit is set to 1 after instruction RORA:

A original \$7C = % 01111100
After RORA = % 10111110_←
Carry Bit

After RORA, A = \$bE and the carry bit of the CCR will be set to 0 since bit-0 was shifted into the C-bit of the CCR.

- E. TAB (Transfer from accumulator A to accumulator B):** This instruction moves the contents of accumulator A to accumulator b. It also sets the sign bit and zero status accordingly, and clears the overflow status. (See page 69 of the M6800 PROGRAMMING MANUAL.)
- F. TBA (Transfer from accumulator B to Accumulator A):** This instruction moves the contents of accumulator b to accumulator A. It also sets the sign bit and zero status accordingly and clears the overflow status. (See page 71 of the M6800 PROGRAMMING MANUAL.)

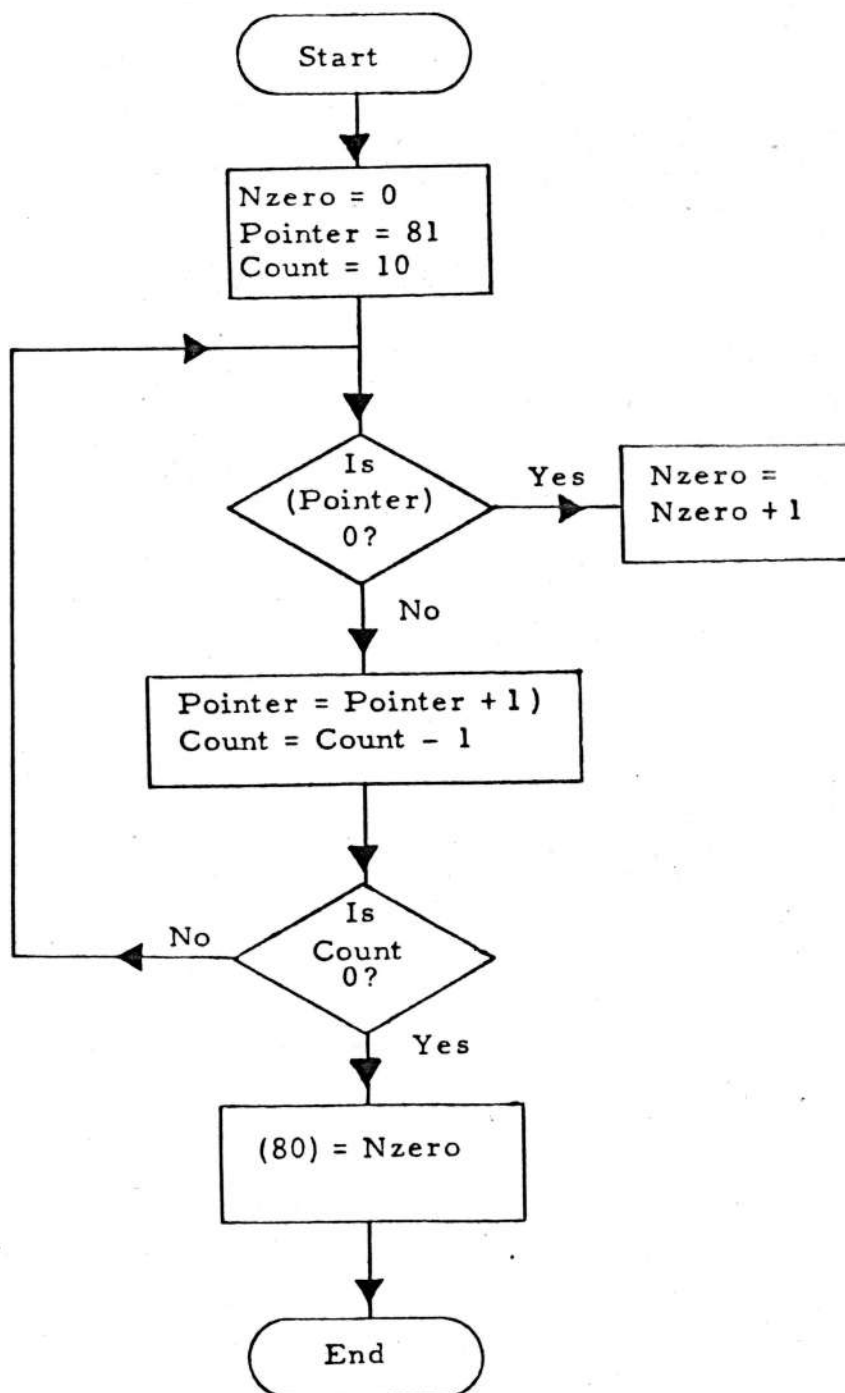
III. Laboratory Exercise.

A. Designing and Debugging Programs on the System X.

1. For the present, we will concentrate on simple, well defined problems which can be designed with flowcharts and can be debugged and tested at the same time. Flowcharting is the traditional program design method and is useful for small problems.

2. Example #1 - Counting zeros: Purpose - count the number of zeros in memory location \$0081 through \$008A and place the result in memory location \$0080. Now let's develop a flowchart as found in Figure 10-1.

Figure 10 - 1
Flowchart for Counting Zeros



3. The next step is to translate the flowchart into a trial program. Make sure that the program includes everything in the flowchart, but do not try to go through it in detail by hand. Use the debugging facilities in the System X instead.

a. As you are aware from Chapter 6, the program may be interrupted at any point by placing a break point or SWI (3F) at any instruction point. You can restart the program by pressing the "DO" key. Not only does the break point save the Program Counter (PC) and the Conditional Code Register (CCR), but all the other registers which you may examine along with any memory location. Remember that the index register, program counter, and the stack pointer are 16-bits long, while the accumulators and the CCR are only 8-bits long.

b. Conditional Code Register (CCR) organization should be reviewed.

Organization of CCR	Bit-7 Always 1
	Bit-6 Always 1
	Bit-5 Half-carry (from bit-3)
	Bit-4 Interrupt (disable)
	Bit-3 Negative (sign)
	Bit-2 Zero
	Bit-1 Overflow
	Bit-0 Carry

Location	Hex	Instr.	Operand # , X	Comment
0 0 6 0	4 F	C L R A		NUMBER OF ZEROS - 0
0 0 6 1	D 6	L D A B	\$ 1 0	COUNT = 10
0 0 6 2	1 0			
0 0 6 3	C E	L D X	# \$ 0 0 8 1	POINTER = START OF ARRAY ZERO
0 0 6 4	0 0			
0 0 6 5	8 1			
0 0 6 6	6 D	T S T		
0 0 6 7	0 0			
0 0 6 8	2 7	B E Q	C O U N T	IS ELEMENT ZERO
0 0 6 9	0 3			
0 0 6 A	5 C	I N C B		YES, ADD 1 TO NUMBER OF ZEROS
0 0 6 B	8 0	I N X		
0 0 6 C	5 A	D E C B		
0 0 6 D	2 6	B N E	Z E R O S	
0 0 6 E	F 4			
0 0 6 F	9 7	S T A A	\$	SAVE NUMBER OF ZEROS(COUNT)
0 0 7 0	8 0			
0 0 7 1	3 F	S W I		

4. Enter the program. Load memory locations \$0081 through \$008A with all zeros. Run the program. Did it work? Examine memory location \$0080. The program, of course, did not work and really for the purpose of this chapter was not supposed to. You could at this point, insert a break point or you could single step through the program. Let's try the latter.

- Depress Set PC
- Load starting address (\$0060)
- Depress "SS" key

5. What are your register contents now? Mine were:

PC/CCR 0061 d4

X/A 0081 00

SP/B 000b 81

a. Do these results look okay to you? They should because the only thing affected by the first instruction will be the A accumulator which will be set to 00 by that instruction.

b. Depress "SS" key.

6. What are your register contents now? Mine were:

PC/CCR 006c d8

X/A 0081 00

SP/B 000b 81

a. It is obvious that the result for the B accumulator is wrong since the instruction at \$0061 is LDAb \$10, and we have some other value in the B accumulator. Examining the instruction we find that d6 is load the B accumulator in the direct mode or from memory location \$10. since we want the value \$10 loaded into the B accumulator immediately, we should use the immediate mode or instruction or LDAb #\$10. However, since we want the decimal number 10 and not hexadecimal number 10, we would have to use the value A (decimal 10), so the correct instruction would be LDAb #\$0A. The instructions should read:

0061 C6 LDAb #\$0A

0062 0A

b. Make this correction by depressing the "examine" key, addressing memory location \$0061, depressing the "change" key, and placing C6 in that location. Now depress the "FWD" key. The display will show \$0062 10. Depress the "change" key and place 0A in that location. Now that the instruction has been corrected, the program may be resumed by depressing the "SS" key. (Remember, the program counter knows where we are.)

7. What are your register contents now? Mine Were:

PC/CCR 0066 d0

X/A 0081 00

SP/B 000b 10

Look strange? The B accumulator still has a 10 in it and we just finished changing the instruction so it would load the value \$0A. You could have some other value in the b accumulator, but the important thing at this point is to remember our PC was sitting at \$0063, which means we would have already completed the instruction at \$0061. The way to correct this would be to depress set PC, address the starting address \$0060, and then depress the "SS" key until the PC shows \$0066. Did the results look okay to you?

8. What are your register contents now? Mine were:

PC/CCR 0066 d0

X/A 0081 00

SP/B 000b 0A

a. Do these look okay to you? They should for the last instruction was LDX#\$0081, and the index register now has the value \$0081, the A accumulator has \$00, and the b accumulator has \$0A.

b. Depress the "SS" key.

9. Examine your register contents again.

PC/CCR 0068 d4

X/A 0081 00

SP/B 000b 0A

a. It is clear that something has happened to the CCR. Let's examine it and see which flag has been affected.

d 4

1 1 0 1 0 1 0 0

Since bits-6 and 7 are always one set, there would be two other bits set to one. They would be bit-2 and bit-4. Since bit-2 is the one that changed, let us examine and see the results of the last instruction, TST \$00,X, which is to test memory location \$0081 since that is what is in the index register. Remember that the TST instruction will only affect flags and will not change the contents of any register, so the CCR tells us that the zero flag is set.

b. Depress the "SS" key.

10. What are your memory contents? Mine were:

PC/CCR 006d d4

X/A 0081 00

SP/B 000b 0A

a. The program counter has jumped from \$0068 to \$006d. If we examine memory location \$0068, we find that we have a BEQ to count, and since our CCR in that instruction told us that the zero flag was set, we would branch forward from the program counter by adding \$03 to the PC. At least our results are uniform — they are really all messed up. It is clear that we jumped over and did not increment A and X, and B is not being decremented. So, if we examine our original program listing in location \$006A along with the flowchart, we state YES, ADD 1 TO NUMBER OF ZEROS, and since the NUMBER OF ZEROS is the A accumulator, that instruction should be changed to INCA or \$4C instead of \$5C. Also, looking at memory location \$006B, we have the value of \$80 for INX, and looking up INX on the instruction set it has a HEX value of 08, so the program should be changed to the following:

006A 4C INCA

006b 08 INX

- b. Obviously, this still does not change the jump instruction, so let's try it by hand. What we want is, if the A accumulator is not zero, skip to memory location \$006b instead of proceeding normally to memory location \$006A. The proper instruction would be:

```
0068    26    BNE COUNT
0069    01
```

Make these changes and let us try the program again, but this time since we feel the program should work, let's set a break point instead of using the SS for the first locations.

- (1) Depress Set PC
- (2) Address \$0060
- (3) Depress BP
- (4) Address \$006d

11. What are your register contents? Mine were:

```
PC/CCR 006d d0
X/A     0082 01
SP/B    000b 09
```

- a. These results look good. We have incremented the X register, incremented the A accumulator and decremented the b accumulator.
- b. Depress the "SS" key.

12. What are your register contents? Mine were:

```
PC/CCR 0063 d0
X/A     0082 01
SP/B    000b 09
```

- a. These results look okay, but let us single step through the program to make sure our jump is okay.
- b. Depress the "SS" key.

13. What are your register contents? Mine were:

```
PC/CCR 0066 d0
X/A     0081 01
SP/B    000b 09
```


- a. Our results show that the X register has decremented, and we do not have an instruction to do that, so something has changed the X. Looking at the branch instruction, we tell the PC to branch to the instruction LDX # \$0081. This means that we would load the X register with \$0081 each time the program was run so the only memory location we would examine would be \$0081. It is obvious that we should jump to location \$0066 which would be the TST 00,X, and then call this location ZEROS, so a new value should be calculated for the relative offset.

Present location of branch instruction	\$006d
Plus 2	\$0002
	\$006F
Location of new instruction	\$0066
Subtract	0009
2's Complement	\$ F7

- b. The value at \$006E should be changed to read \$F7.

14. Now try running the program without break points. Are the results correct? Test the program with other data that is not all zeros.

15. Your final program should be:

Location	Hex	Instr	Operand # X	Comment
0,0,6,0	4F	CLRA		NUMBER OF ZEROS = 0
0,0,6,1	C6	LDA B	# \$0,A	COUNT = 10
0,0,6,2	0A			
0,0,6,3	CE	LDX	# \$0,0,8,1	POINTER. START OF ARRAY
0,0,6,4	00			
0,0,6,5	81			
0,0,6,6	6D	TST X		EXAMINE AN ELEMENT
0,0,6,7	00			
0,0,6,8	26	BNE	COUNT	IS ELEMENT ZERO
0,0,6,9	01			
0,0,6,A	4C	INCA		YES, ADD 1 TO NUMBER
0,0,6,B	08	INX		COUNT
0,0,6,C	5A	DECB		
0,0,6,D	26	BNE	ZEROS	
0,0,6,E	F7			
0,0,6,F	97	STA A		SAVE NUMBER OF ZEROS
0,0,7,0	80			
0,0,7,1	8F	SWI		

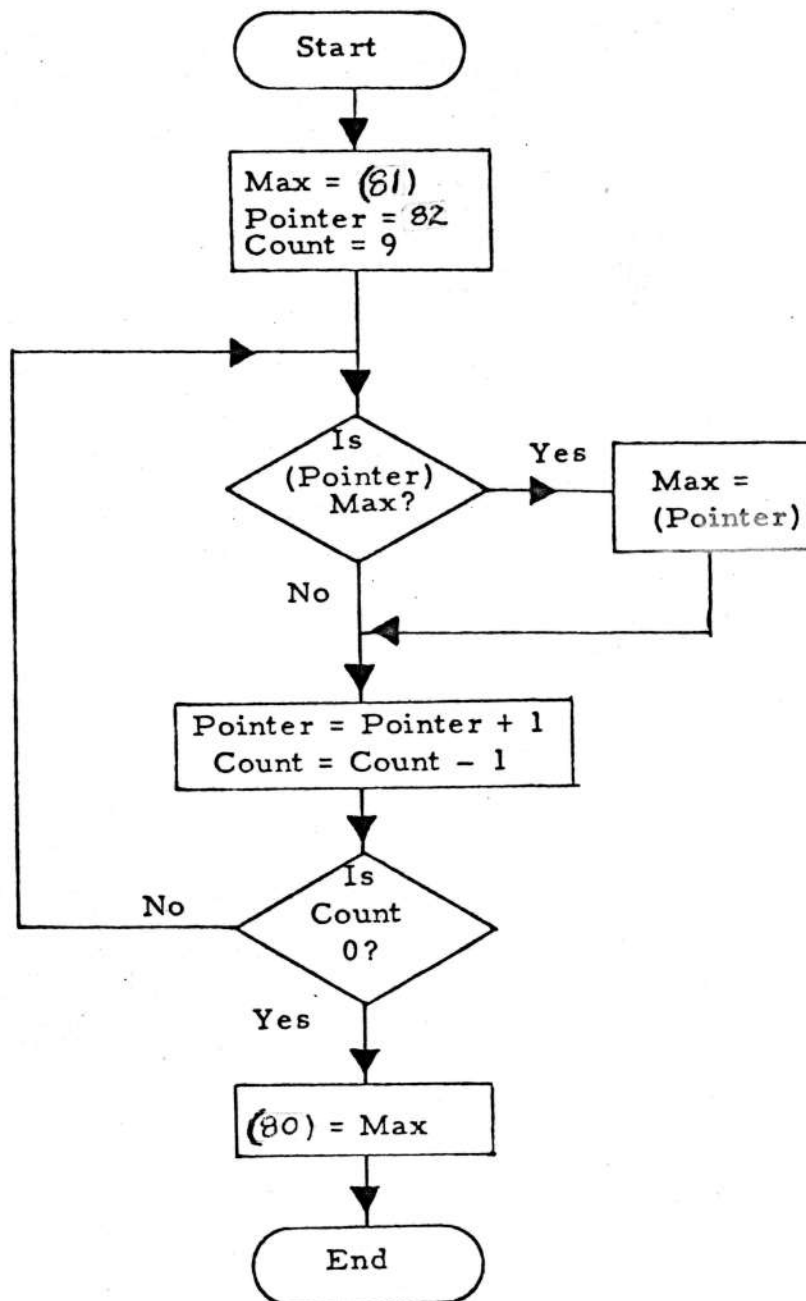
B. Common Errors

1. Inadvertantly reinitializing a memory location or register by incorrect branching.
2. Incorrect branching (branching on **not equal** rather than **equal**).
3. Failing to update a pointer or counter. Be aware of loop controls to be updated regardless of the program path.
4. Failing to initialize variables. Assume nothing !!!
5. Wrong entries into the keyboard. Always examine your program before running it.
6. Incorrect calculation of relative offsets. Don't forget to start at the instruction following the branch.
7. Confusing data and addresses. The memory location can contain anything, even the number 10.
8. Mixing decimal and hexadecimal numbers. Decimal 10 is 0A HEX, and HEX 10 is decimal 16.
9. Mixing the index register with the address in the index register. STAA,X stores accumulator A to the address in the index register.

- C. Example #2 — Find Maximum: Purpose —** find the largest unsigned number in memory locations \$0081 through \$008A and place it in memory location \$0080. Follow the flowchart Figure 10-2.

Note: You do not have to make the flowchart very elaborate or very detailed. In fact, the flowchart is most useful when it is a simple, straightforward, guide to the logic of the program.

Figure 10 - 2
Flowchart for Find Maximum



D. Arithmetic on the System X

1. One common task for small computers is simple arithmetic. Typical applications involving arithmetic include: averaging sets of readings, forming checksums, comparing levels to thresholds, scaling inputs and outputs, linearizing non-linear inputs (such as thermocouples), and calculating frequency responses. Decimal arithmetic is necessary in calculators, business terminals, instruments, appliances, and games.
2. An earlier exercise showed how to form an 8-bit sum. We will now extend that exercise to saving carries, and forming a 16-bit sum; performing simple multiplication, division, and rounding; and handling multi-word binary and decimal arithmetic.

E. An 8-Bit Sum

1. Purpose - add together the array of elements starting in memory location 83 and place the sum in memory location 82 (ignoring carries). The length of the array is memory location 80.

Location	Hex	Instr.	Operand # , X	Comment
0,0,6 ⁰	4 F	C L R A		SET SUM TO ZERO
0,0,6 ¹	C E	L D X	# \$, 8, 3	POINTER START OF ARRAY
0,0,6 ²	0,0			
0,0,6 ³	8,3			
0,0,6 ⁴	A B	A D D A	0,0, X	SUM = SUM + DATA ADD
0,0,6 ⁵	0,0			
0,0,6 ⁶	0,1	N O P		
0,0,6 ⁷	0,8	I N X		
0,0,6 ⁸	7, A	D E C	\$, 0, 0, 8, 0	DECREMENT COUNTER
0,0,6 ⁹	0,0			
0,0,6 ^A	8,0			
0,0,6 ^B	2,6	B N E	A, D, D	
0,0,6 ^C	F,7			
0,0,6 ^D	9,7	S T, A A		STORE SUM DONE
0,0,6 ^E	8,2			
0,0,6 ^F	3, F			

2. Enter this program and run it. What happens if (80) = 0? How would you correct the program to solve this problem:

Sample Data: (80) = 02
(83) = 6E
(84) = 39

Result: (82) = A9

F. Decimal Arithmetic

1. The instruction DAA (Decimal Adjust accumulator A) corrects a binary sum in accumulator A to a decimal sum, (i. e. the two instructions perform a decimal addition). Note that you cannot decimal adjust accumulator B.

ADDA
DAA

2. Try the earlier 8-bit addition program with and without the DAA instruction:

	CLRA	83	SUM = 0
	LDX	#\$	POINT TO START OF ARRAY
AddW	AddA	X	SUM = SUM + DATA
	(NOP)	(DAA)	DECIMAL OR BINARY SUM
	INX		
	DEC	\$80	
	BNE	AddW	
	STAA	\$82	
	SWI		

3. Replace the 01 in memory location 0066 with DAA (19). Use the following set of data:

(90) = 03
(83) = 16
(84) = 26
(85) = 35

Result: (82) = 77 (with DAA)
= 71 (without DAA)

4. What is the reason for this difference?
5. You can easily save the carries by placing them in accumulator B.

A useful instruction is:

ADCB #0

The result is $(b) = (b) + \text{CARRY} + 0$
 $= (b) + \text{CARRY}$

6. Use this instruction to save the carries from the 8-bit addition. Store the carries in memory location 81.

- a. Run the revised program with the following data:

(80) = 04

(83) = bF

(84) = 78

(85) = E1

(86) = F1

Result: (81) = 03
(82) = 09

- b. Try a decimal version with the following data:

(80) = 0C

(83) = 93

(84) = 88

(85) = 98

(86) = 97

(87) = 94

(88) = 92

(89) = 90

(8A) = 97

(8B) = 93

(8C) = 96

(8D) = 95

(8E) = 97

Result: (81) = 11
(82) = 30

7. Remember to decimal adjust both the sum and the carries. But remember that DAA only works on accumulator A. You can use the following instruction sequence to produce a 16-bit decimal sum:


```

AddW AddA X      SUM = SUM + DATA
      DAA        DECIMAL SUM
      STAA
      $82 TBA
      ADCA#0     ADD IN CARRIES
      DAA        AND MAKE THAT DECIMAL
      TAB
      LDAA$82

```

8. The various codes are:

```

TBA      17
ADCA #   89
TAB      16

```

G. Performing 16-Bit Arithmetic

1. You can use the two accumulators together to perform 16-bit arithmetic. The following program will add a set of 16-bit numbers stored starting in memory location 83 (least significant bits first). The length of the set is in memory location 80.

Location	Hex	Instr.	Operand #	X	Comment
0 0 6 0	4 F	CLR A			SET SUM TO ZERO
0 0 6 1	5 F	CLR B			
0 0 6 2	C E	L DX	#	\$ 0 0 8 3	POINTER START OF ARRAY
0 0 6 3	0 0				
0 0 6 4	8 3				
0 0 6 5	A B	ADD A		0 0, X	ADD 8 LSBs OF DATA. ADD
0 0 6 6	0 0				
0 0 6 7	0 8	INX			
0 0 6 8	E 9	ADC B		0 0, X	ADD 8 MSBs OF DATA
0 0 6 9	0 0				
0 0 6 A	0 8	INX			
0 0 6 B	7 A	DEC		\$ 0 0 8 0	
0 0 6 C	0 0				
0 0 6 D	8 0				
0 0 6 E	2 6	BNE		A D D	
0 0 6 F	F 5				
0 0 7 0	D 7	STA B			STORE MSBs OF SUM
0 0 7 1	8 1				
0 0 7 2	9 7	STA A			STORE LSBs OF SUM
0 0 7 3	8 2				
0 0 7 4	3 F	SWI			

2. Enter this program and try it on the following data:

(80) = 03
(83) = F8
(84) = 37
(85) = 19
(86) = 26
(87) = EC
(88) = 0b

Result: 37F8
 + 2619
 + 0BEC
 69FD

(81) = Fd
(82) = 69

3. Change the program so that it performs a 16-bit logical sum (Exclusive OR).
4. How about a 16-bit logical AND.
5. Revise the program so that it uses a final zero element, rather than a specific count.
6. Try the last program on the following sets of data:

(83) = 00
(84) = 00

Result: (81) = 00
 (82) = 00

(83) = 67
(84) = 02
(85) = 80
(86) = 80
(87) = 00
(88) = 00

Result: (81) = 81
 (82) = 82

7. Remember that $X + Y = 0$ does not mean either X or Y is zero (why?).
8. How can you determine if a 16-bit number is zero?

H. Rounding Binary Numbers

1. Rounding binary numbers to a specified bit length is simple since the only bit values are 0 and 1. All you have to do is look at the most significant bit that you plan to drop.
 - a. If MSB = 1, round up by adding 1.
 - b. If MSB = 0, round down by truncating.
2. The following program will round a 16-bit number in memory locations 81 and 82 (MSB's in 81) to an 8-bit number in memory location 80:

Location	Hex	Instr.	Operand #	X	Comment
0 0, 6 ⁰	9 6	L D A	A	\$ 8 1	GET MSBs
0 0, 6 ¹	8 1				
0 0, 6 ²	D 6	L D A	B	\$ 8 2	GET LSB
0 0, 6 ³	8 2				
0 0, 6 ⁴	2 A	B P L		S T O R R	DO LSBs NEED ROUNDING
0 0, 6 ⁵	0 1				
0 0, 6 ⁶	4 C	I N C	A		YES, ADD 1 TO MSBs
0 0, 6 ⁷	9 7	S T A A	# 8, 0		STOR
0 0, 6 ⁸	8 0				
0 0, 6 ⁹	3 F	S W I			

Try this program on these sets of data:

(81) = 26

(82) = 88

Result: 80 = 27

(81) = 43

(82) = 7F

Result: (80) = 43

3. You can scale a number down by dividing it by 2, (i. e. you can truncate an 8-bit number to 7-bits with the single instruction LSR). You can shift both accumulators right by using the logical shift on the MSBs and the rotate instruction on the LSbs. A left shift uses the opposite sequence.

a. 16 — BIT RIGHT SHIFT A AND b
(MSBs IN A)

LSRA
RORB

b. 16 — BIT LEFT SHIFT A AND b
(MSBs IN A)

ASLB
ROLA

4. Similar sequences will work on two consecutive memory locations:

LSR \$82
ROL \$83

5. The following program will scale the contents of memory locations 82 and 83 by a factor of 2, round the result, and save it in memory locations 80 and 81 (MSBs in 80 and 82).

Location	Hex	Instr.		Operand # , X	Comment
0,0,6 ⁰	9,6	L,D,A	A	\$,8,2	GET MSBs
0,0,6 ¹	8,2				
0,0,6 ²	D,6	L,D,A	B	\$,8,3	GET LSBs
0,0,6 ³	8,3				
0,0,6 ⁴	4,4	L,S,R	A		SCALE DOWN BY 2
0,0,6 ⁵	5,6	R,O,R	B		
0,0,6 ⁶	C,9	A,D,C	B	#,0,0	ROUND LSBs
0,0,6 ⁷	0,0				
0,0,6 ⁸	8,9	A,D,C	A	#,0,0	ROUND MSBs
0,0,6 ⁹	0,0				
0,0,6 ^A	9,7	S,T,A	A		STORE MSBs
0,0,6 ^B	8,0				
0,0,6 ^C	D,7	S,T,A	B		STORE LSBs
0,0,6 ^D	8,1				
0,0,6 ^E	3,F	S,W,I			

6. Try this program on the following data:

(82) = 57

(83) = 83

Result: (80) = 2b
(81) = C2

(82) = 16

(83) = 80

Result: (80) = 0b
(81) = 40

7. Change the program so that it scales the original number by a factor of 4 and rounds. The results should be:

(82) = 57

(83) = 83

Result: (80) = 15
(81) = E1

(82) = 16

(83) = 80

Result: (80) = 05
(81) = A0

8. Now change the program so that the number of bits to be dropped (0) is in memory location 84:

(82) = 57

(83) = 83

(84) = 01

Result: (80) = 2b
(81) = C2

(82) = 57

(83) = 83

(84) = 03

Result: (80) = 0A
(81) = F0

Chapter 11: Monitor Programs and the ASCI T68 Bug

Objective: The student should become familiar with the Programmable Read Only Memory (PROM) subroutines, using the stack, the ASCI System X T68 bug monitor, and how to use the T68 bug monitor for user programs.

Student study requirements: The USER'S MANUAL; ASCI T68 bug monitor program printout and subroutine explanation; and page A57 of the M6800 PROGRAMMING MANUAL.

Discussion

I. General Information on Subroutines

As mentioned in previous chapters, a branch may be either in the form of a jump or a subroutine call. The subroutine is basically a series of instructions that are used repeatedly in the course of running the major program. Its main reason for existence is to save program steps and to shorten the program execution time.

The step saving character of a subroutine requires some explanation. Suppose a routine is to be written to multiply a sequence of several numbers. Multiplication can be accomplished by a method of repeated addition. This method requires repeatedly adding a number to itself the number of times specified by a second number. A routine to accomplish the multiplication of two numbers can be written and,

depending on the numbers, may be twenty or thirty steps long. The problem here, however, is to multiply not just two numbers, but a sequence of numbers. If there were ten numbers to be multiplied, the program could be two or three hundred steps long. By using the subroutine idea, a program could be written to get the sequence of numbers located properly for the call for the subroutine. In this way, the multiply routine needs to be written only once and then called on ten times, resulting in a saving of two hundred to two hundred and fifty steps.

Just from this basic idea you should have the idea that jump and subroutine call instructions provide a tremendous decision making power. The implementation of the individual concept block depends on the decisions required and the ability of the programmer to create the branch conditions.

The use of subroutines also offers the programmer certain design freedoms not available in straight line programming. That is, he can design subroutines as separate modules or utility routines and call on them as they are needed in the main program operation. This approach does require that the subroutines be relatively independent and adhere to the "think of everything" philosophy. The modular programming approach is a sophisticated method and requires background experience beyond the beginner level, as well as a thorough understanding of the capabilities of the machine involved. However, at this point in your instruction on the System X, you should be capable of writing some basic subroutines and basic monitor programs.

II. New Instructions

RTS (Return from Subroutine): After execution of a BSR or JSR and the completion of the subroutine that the MPU branched to, this instruction will return to the main program at the original address plus two if the instruction was a BRANCH, or to the original address plus three if the instruction was a JUMP. (See page A57 of the M6800 PROGRAMMING MANUAL.) For a good example of how the RTS is used, refer to the ASCII T68 bug printout. Notice at the end of one of the subroutines, you will find the RTS (HEX code 39).

III. Laboratory Exercise

A. Subroutines and the ASCII System X Monitor

1. Clearly some instruction sequences will be used over and over again, e.g., the delay routine, keyboard routine, display output, etc. You will probably find it convenient to write these routines once, place them in memory, and simply refer to them as needed.
2. You may call the subroutine with the instructions BSR or JSR. These instructions perform an unconditional jump to the start of the subroutine and save the old value of the program counter in memory. An RTS instruction at the end of the subroutine restores the old program counter and returns control to the main program.

B. A Delay Subroutine

1. For example, the one second delay routine could serve as a subroutine. The following program waits for key "C" to be pressed and then delays for one second:

Location	Hex	Instr.	Operand # , X	Comment
00,60	b6	L D A A	E 0 0 2	GET KEYS "8" - "F" WAIT
00,61	E0			
00,62	02			
00,63	84	A N D A	# % 1 0	IS KEY "C" PRESSED
00,64	10			
00,65	26	B N E		NOT EQUAL. GO TO WAIT
00,66	F0			
00,67	bd	J S R	0 0 9 0	GO TO 1-SEC. DELAY
00,68	00			
00,69	90			
00,6A	3F	S W I		
	B			
	C			
	D			
	E			
	F			
00,90	CE	L D X	# F 4 2 3	DELAY 1-SEC. DLY1
00,91	F4			
00,92	23			
00,93	09	D E X		
00,94	26	B N E		NOT EQUAL. GO TO DLY1
00,95	Fd			
00,96	CE	L D X	# F 4 2 3	DLY2
00,97	F4			
00,98	23			
00,99	09	D E X		
00,9A	26	B N E		NOT EQUAL. GO TO DLY2
00,9B	Fd			
00,9C	39	R T S		

2. Enter the program and run it. Remember you must depress the "C" key and wait one second.
3. Now let's run the program again. Depress "SET PC", address 0060, depress BP, and address 0099. Depress the "C" key. In effect we are going to set a break point at \$0099 and look at the effect of JSR on the stack. Examine your SP/B. Notice that the stack pointer is located at address 0009. Depress the "examine" key and address 0009. Since the stack pointer is located at this location, depress the "F" key and look at the last data placed in the stack which will be the CCR, so the stack will be loaded as follows:

ADDRESS	DATA
000A	CCR
000B	B accumulator
000C	A accumulator
000D	Index (high byte)
000E	Index (low byte)
000F	Program Counter (high byte)
0010	Program Counter (low byte)

4. The reason for the change is that JSR used the stack pointer to store data in memory. The procedure for each byte is:
 $((\text{Stack Pointer})) = \text{Data}$
 $(\text{Stack Pointer}) = (\text{Stack Pointer}) - 1$
5. Each byte goes into the address contained in the stack pointer and that address is decremented so the next byte will go into the next lower address. The stack grows downward so that you can start it at the end of a block of memory and it will not interfere with other data.

6. The ASCII T68 bug monitor maintains separate stack areas for user and monitor operations. On power up or RESET, a ten byte area of RAM (\$0000-\$000b) is allocated for the user's stack.

Note: (This area is sufficient to run and debug user programs with one level of subroutines only.) Normally programming practice would require the user program to initialize its own stack area. The instruction LDS (Load Stack Pointer) will accomplish this. (See page A46 of the M6800 PROGRAMMING MANUAL.) You may also refer to page 112 of BASIC MICRO-PROCESSORS and THE 6800 by Bishop.

7. The instructions RTS and RTI use the stack pointer to retrieve data and addresses from memory. The procedure for each byte is:

$$(\text{Stack Pointer}) = (\text{Stack Pointer} + 1)$$
$$\text{Data} = ((\text{Stack Pointer}))$$

The address in the stack pointer is incremented before a byte of data is retrieved. The next byte will be obtained from the next higher address. So RTS and RTI reverse the actions of JSR,BSR, and SWI.

C. A Keyboard Subroutine

1. The following subroutine waits for one of keys "0" to "7" to be pressed and returns with the key identification in accumulator B.

Location	Hex	Instr.	Operand # , X	Comment
0				
1				
2				
3				
4				
5				
6				
00,97	b6	LDA	A E,0,0,0	GET KEYS "0" - "7" WAIT
00,98	E0			
00,99	00			
00,9A	81	CMP	A #,\$FF	ARE ANY KEYS PRESSED?
00,9B	FF			
00,9C	27	BEQ		IF EQUAL, GO TO WAIT
00,9D	F9			
00,9E	5F	CLR	B	YES KEY NUMBER = 0
00,9F	44	LSR	A	IS NEXT BIT 0? SV KEY
00,A0	24	BCC		YES DONE
00,A1	03			
00,A2	5C	INC	B	NO, ADD 1 TO KEY NUMBER
00,A3	20	BRA		GO TO SV KEY
00,A4	FA			
00,A5	39	RTS		DONE

2. Revise the earlier program combined with the above program so that it waits for key "C", then waits until one of keys "0" to "7" is pressed, and then delays for the correct number of seconds. Place a break point at memory location \$009A and see what happens to the stack.

D. A Display Subroutine

1. The following program converts the contents of accumulator B to a seven-segment code and sends the results to the LEDs. The original contents of accumulator B are unchanged.

Location	Hex	Instr.	Operand # , X	Comment
0				
1				
2				
3				
4				
5				
00A6	86	L D A A	# \$FF	GET BASE PAGE OF 7-SEG. CODE
00A7	FF			
00A8	97	S T A A	\$8E	
00A9	8E			
00AA	d7	S T A B	\$8F	INDEX TABLE WITH DATA
00AB	8F			
00AC	dE	L D Y	# \$8E	GET CODE ADDRESS
00AD	8E			
00AE	A6	L D A A	,0,d,X	GET CODE
00AF	0d			
00b0	b7	S T A A	E402	GET LEDs
00b1	E4			
00b2	02			
00b3	39	R T S		

2. Remember that the table of seven-segment codes starts in memory location FF0D. Note that the subroutine uses the A accumulator, the index register, and memory locations \$008E and \$008F.
3. Revise the earlier program so that it displays the number of seconds remaining. How would you revise this combination of subroutines so that the "F" key turns the displays on and off,

i.e., pressing the "F" key once turns the displays on, and pressing it again turns the displays off. This feature is often convenient when the displays may be distracting to an operator.

E. Using the ASCI T68 Bug Monitor Subroutines

1. You can also use the JMP and JSR instructions to reach the subroutines in the ASCI System X. These subroutines are fully defined in the USER'S MANUAL. The ASCI T68 bug gives you two methods of getting to the monitor subroutines. You may either use the JMP instruction to a jump vector, or the JSR to the memory location of the subroutine.
2. The following list will give the most commonly used subroutine labels, their jump vectors, and memory locations. For detailed explanations see the USER'S MANUAL starting on page 12, and the ASCI T68 bug printout.

SUBROUTINE LABEL	JUMP VECTOR	MEMORY LOCATION
DISPLY	F80C	FD74
WRDATA	F80F	FE22
KEYIN	F803	FD52
OUT2HX	No JMP Vector	FD27
ADROUT	F812	FD1B
IN2HX	F806	FD38
INEE	F81B	FC46
INHEX	F81E	FC33
IN2	F821	FC22
IN4	F824	FC20
OUTEE	F827	FC6E
OUTHEX	F82A	FC66
OUT2	F82D	FC55
OUT4	F830	FC53
BAUD	F833	FCB9

3. Remember the DISPLY only scans the LEDs once. The following program will hold the contents of memory locations \$0080 through \$0085 on the LEDs. Remember that we have to move our data to the display locations since the monitor uses the display locations also.

Location	Hex	Instr.	Operand #, X	Comment
0 0 6 0	7 F	CLR	0 0 6 d	CLEAR FETCH START
0 0 6 1	0 0			
0 0 6 2	6 d			
0 0 6 3	C 6	LDX	0 0 8 0	DATA BEGIN
0 0 6 4	0 0			
0 0 6 5	8 0			
0 0 6 6	C 6	LDAB #	\$ 0 6	
0 0 6 7	0 6			NMBR
0 0 6 8	8 6	LDAA #	\$ 4 A	
0 0 6 9	4 A			DIGIT STORAGE
0 0 6 A	9 7	STAA	\$ 6 f	
0 0 6 B	6 F			
0 0 6 C	A 6	LDAX	0 0 X	LOAD DATA
0 0 6 D	0 0			FETCH POINTER
0 0 6 E	9 7	STAA		
0 0 6 F	0 0			COUNTER POINTER
0 0 7 0	7 C	INC	0 0 6 d	INC FETCH
0 0 7 1	0 0			
0 0 7 2	6 d			
0 0 7 3	7 C	INC	0 0 6 F	INC COUNTER
0 0 7 4	0 0			
0 0 7 5	6 F			
0 0 7 6	5 A	DECB		DECREMENT NMBR
0 0 7 7	2 6	BNE		NOT 0 GO TO LOAD DATA
0 0 7 8	F 3			
0 0 7 9	b d	JSR		GO TO DISPLAY SCAN
0 0 7 A	F d			
0 0 7 B	7 4			
0 0 7 C	2 0	BRA		GO TO SCAN
0 0 7 D	F 5			

4. Try the following patterns:

a. \$80 = 6E

\$81 = 9E

\$82 = 1C

\$83 = 1C

\$84 = FC

\$85 = 00

b. \$80 = 00

\$81 = 8E

\$82 = 1C

\$83 = FC

\$84 = CE

\$85 = 00

5. Make some patterns of your own and display them.

6. A good example of the use of subroutines is demonstrated in the "Hello Can I Help You" program in Chapter 3, page 22 of this manual.

a. Turn to page 24 of this manual — memory location \$00A7. Notice the bd F80C or JSR to DISPLY.

b. You can start at memory address \$0089, single step through this program, and see how the data is loaded for display. Give a written explanation of how this happens in this program.

F. Building Monitor Commands

1. Since you have been using some routines in this chapter that would serve as basic monitor routines at this time, using the programs just completed, write a program that will look at any of the 16 keys, and then display the first six keys depressed. For example, if you depressed 0, 1, 2, 3, A, B, then the LEDs would show 0123 AB. You will have to rearrange some of the previous programs as they are located in the same memory locations. However, this will provide you with the experience of proper setup of subroutines.
2. If the above question does not provide you with enough challenge, then try the following:

- a. Write a basic monitor routine to build an address from four-key entries and display the contents of that address.

JSR	Get first two digits of address
STAA	Address holder upper
JSR	Get second two digits of address
STAA	Address holder lower
LDX	Address holder, make into address
LDAA,X	Get data from address
JSR	Data to display buffer
JSR	Display
BRA	Display

- b. Be sure to review those subroutines in the ASCII T68 bug monitor. They should provide you with ideas so you may either write your own or use those subroutines in the ASCII T68 bug. Either way, this will provide you with an appreciation of monitor subroutines and how to use these subroutines to your advantage as a programmer of machine language.

Chapter 12: The Peripheral Interface Adapter (PIA) and the Asynchronous Communication Interface Adapter (ACIA)

Objective: To understand the ASCI System X PIA and ACIA ports, Control Register, Data Direction Register, General and Parallel Addressing, and the PIA and ACIA control lines.

Student study requirements: Pages 164 through 207 of BASIC MICRO-PROCESSORS and THE 6800 by Bishop; and pages 2-4 to 2-6 of the M6800 PROGRAMMING MANUAL.

Discussion

I. Introduction to the Peripheral Interface Adapter (PIA)

The Peripheral Interface Adapter, which we will regularly call the PIA, is a microprogrammed universal parallel interface. It provides connection to the outside world via two 8-bit bidirectional buses and four control lines. The biggest problem the user has with this device is understanding the effect of each of the bits in the control logic of the PIA which he must properly set up before he can expect it to function in his system. The PIA appears messy because it is a universal device — a device which is to be all things to all people.

The drawing in Figure 9-14 on page 165 of BASIC MICROPROCESSORS and THE 6800 by Bishop, gives a basic picture of the internal registers of the PIA. Note that it has six 8-bit registers. These are organized in two nearly identical groups, designated the A side and the B side. We are going to confine ourselves in this description entirely to the A side. The B side differs only in a couple of very tiny details which you are going to notice only when you study the interface description in Motorola's manual very carefully.

The three registers are named the Data Direction Register, the Output Register, and the Control Register. To avoid writing out these names, particularly where space is at a premium, we will use their first letters, adding an A or a B to designate which side of the PIA we are dealing with. Hence, the Data Direction Register on the A side becomes DDRA, the Output Register on that side becomes ORA, and the Control Register becomes CRA. Look at the A side of the PIA on Figure 9-14, page 165 of BASIC MICROPROCESSORS and THE 6800 by Bishop, and we will describe each of the registers.

The data direction register assigns the direction of each of the bidirectional lines of the 8-bit bus to the outside world. The concept is very simple. There is a bit in the DDR corresponding to each line. If a given line is to be used for output, that is, if it is to provide a signal for use by external circuitry, the corresponding bit in the DDR must be set to 1. If the given line is to be used for input, the corresponding bit must be set to 0. For example, if we design a particular external circuit which requires that line PA3 be used for output, the DDRA bit-3 must be 1. Note that this setup is likely to be done only once at the beginning of a program, because we are likely, at least in simpler input/output circuits, to have a single assignment, either input or output, for each peripheral data line.

The output register has eight bit positions which correspond to the eight peripheral data lines. The program can place a byte in the output register. When it does, those peripheral data lines which are conditioned for output by the data direction register will be set to levels corresponding to the bit pattern in the output register. Note that the DDR must have a 1 in the proper bit position to condition a peripheral data line for output. A bit value of 1 in the output register will then cause that line to be high, while a 0 will cause it to be low. Note further that a 0-bit in the DDR conditions a line to be input, and the bit in the output register has no effect. We will demonstrate this by an example later.

When the data direction register has a line conditioned for input, a high level on the line is interpreted as a 1. This 1 is sent to the buffer for use by the MPU, independent of the contents of the output register. While this may appear to be complex, really it provides considerable flexibility, for the user does not have to worry about keeping input and output lines separate. The PIA does this for him. We will be sloppy, though, and refer to both input and output through the output register, since the same address accesses both the output register and the data buffer. While most applications do not use both directions on a single peripheral data line, complex systems can require this flexibility. Do not be surprised if you have more difficulty understanding the PIA than other parts of the M6800 family. It is by far the trickiest device that we will study.

The control register controls two peripheral control lines and some internal functions of the PIA. Each of the eight bits in the register is assigned a special meaning, and these are generally set to fixed values at the beginning of the program. During execution of a program some of these bits may be changed or examined. Watch out that you understand that you must always deal in bytes! You can change a single bit in this register only by

sending a full byte to it. That byte will have exactly the same bit pattern as was in the control register, with the exception of the bit you want to change. Even so, certain bits of this control register cannot be changed by writing into this register.

We will describe the bits of the control register only in general here and leave the details until later when we have a problem to handle. Note that here as well as everywhere else in this course, we will be numbering the bits from right to left, starting at 0. Hence, the right most bit is bit-0, and the left most bit in an eight bit register is bit-7:

CRA bit-7	Interrupt flag from line CA1
CRA bit-6	Interrupt flag from line CA2 when CA2 is conditioned for input
CRA bits-5,4,3	Conditioning of line CA2
CRA bit-2	Switch between ORA and DDRA: If this bit is 1, data from MPU goes to ORA; if it is 0, data from MPU goes to DDRA.
CRA bits-1,0	Conditioning of line CA1

How do we address the registers of the PIA when using them in our programs? Recall the common bus structure of the M6800 system. All system activity is done on this bus using addresses to handle not only the memory locations, but also the other registers, devices, and so on which are placed on that bus. Hence, each of the registers of the PIA has an address. These addresses are assigned by the manner in which the PIA is connected to the address bus. The addresses given in Table 12-1 are for the system which you are going to be using for your programs. Other systems are likely to be somewhat different. Your system has three PIA's, two for the keyboard/display module, and one for you to use for additional experimentation.

Table 12-1
PIA Addresses

Keyboard PIA		Display PIA	Spare PIA
DDRA	E000	E400	E800
ORA	E000	E400	E800
CRA	E001	E401	E801
DDRB	E002	E402	E802
ORB	E002	E402	E802
CRB	E003	E403	E803

Why do the data direction register and the output register have the same address? Motorola apparently made the choice to simplify the address logic of the PIA. Remember that bit-2 of the CRA determines which register is going to be addressed. If bit-2 of the CRA is set to 1, then address E000 refers to the user PIA's output register A; if it is 0, that same address refers to the data direction register A.

The PIA must be initialized to make the various input/output lines correspond to what the external hardware expects. This means that the bits of the data direction register and the control register must be set to appropriate values. This is a tricky job at times and often requires a substantial amount of both thought and program to make it correct. The usual sequence is as follows:

- Set bit-2 of the control register to 0 (via the CLR instruction) to gain access to the data direction register.
- Set the bits of the data direction register for each peripheral data line to designate input or output.
- Set bit-2 of the control register to 1 to gain access to the output register and set the rest of the bits of the control register to properly condition the control lines.

II. Asynchronous Communications Interface Adapter (ACIA)

The Asynchronous Communications Interface Adapter, which is more easily called the ACIA, is a universal asynchronous serial interface. Like the PIA, it is microprogrammed. Many of you will recognize this device as a UART. While it has many features, two should be mentioned here. It handles either an 8-bit or a 9-bit serial transmission with either one of two stop bits. It handles parity generation and checking. The system which you will use for the laboratories contains an ACIA, but this is connected entirely to external circuitry for transmitting data from or to the digital cassette recorder, CRT, and other peripheral equipment. For additional study on the ACIA, see pages 184 through 200 of BASIC MICROPROCESSORS and THE 6800 by Bishop.

III. Laboratory Exercise

A. Spare PIA

1. The ASCI System X has a spare PIA available for your control.

Its addresses are as follows:

Title	Address	Mnemonic
Data Direction Register A	E800	DDRA
Control Register A	E801	CRA
Data Direction Register B	E802	DDRB
Control Register B	E803	CRB

Table 12-2
Input/Output

The input/output for the spare PIA is the 26 pin card edge connector located in the rear of the ASCI System X. If you have an ASCI LS/ICM industrial control module, all I/O are brought out on this module. If you have a 26 pin card edge connector, Table 12-3 gives the pin connections for a standard 26 pin connector.

Table 12-3
Connections for 26 Pin Connector

Title	Pin Number
PA0	20
PA1	19
PA2	18
PA3	17
PA4	16
PA5	15
PA6	14
PA7	13
CA1	12
CA2	11
PB0	10
PB1	9
PB2	8
PB3	7
PB4	6
PB5	5
PB6	4
PB7	3
CB1	2
CB2	1

Pins 21 to 26 are ground

2. The following program makes an input port of the A side:

```

CLR      CRA
CLR      DDRA      MAKE ALL LINES INPUTS
LDAA     #00000100 ACCESS DATA REGISTER
STAA     CRA
  
```


3. The data direction register and the A side data are placed in registers A and B respectively in the program below, for simultaneous viewing.

Location	Hex	Instr.	Operand # , X	Comment
0 0 7 0	7 F	C L R	E 8 0 1	CLR CRA
0 0 7 1	E 8			
0 0 7 2	0 1			
0 0 7 3	7 F	C L R	E 8 0 0	CLR DDRA
0 0 7 4	E 8			
0 0 7 5	0 0			
0 0 7 6	F 6	L D A B	E 8 0 0	LDAB DDRA
0 0 7 7	E 8			
0 0 7 8	0 0			
0 0 7 9	8 6	L D A A	# % 0 4	
0 0 7 A	0 4			
0 0 7 B	b 7	S T A A	E 8 0 1	STAA CRA
0 0 7 C	E 8			
0 0 7 D	0 1			
0 0 7 E	b 6	L D A A	E 8 0 0	LDAA DDRA
0 0 7 F	E 8			
0 0 8 0	0 0			
0 0 8 1	3 F	S W I		

4. After running the program, notice the end values. The values of the data and data direction registers were obtained from the same address. The same program should be run using the PIA B side.
5. One end of an SPST (Single Pole, Single Throw) switch is to be attached to data bit PA7, and ground the other end. This may be accomplished by use of the ASCII LS / ICM where PA7 is located, or use a 26 pin edge connector to the spare PIA output, and connect from pin 13 to pin 26. The program below makes an input port of the PIA and waits for switch closure.

Location	Hex	Instr.	Operand # , X	Comment
0 0 7 0	7 F	C L R	C R A	
0 0 7 1	E 8			
0 0 7 2	0 1			
0 0 7 3	7 F	C L R	D D R A	ALL LINES INPUTS
0 0 7 4	E 8			
0 0 7 5	0 0			
0 0 7 6	8 6	L D A A	# % 0 4	ACCESS DATA REGISTER
0 0 7 7	0 4			
0 0 7 8	b 7	S T A A	C R A	
0 0 7 9	E 8			
0 0 7 A	0 1			
0 0 7 B	b 6	L D A A	D D R A	GET SWITCH WAITS
0 0 7 C	E 8			
0 0 7 D	0 0			
0 0 7 E	2 b	B M I	W A I T S	WAIT TO OPEN
0 0 7 F	F b			
0 0 8 0	3 F	S W I		

6. Clearing the control register permits the placing of values in the data direction register. Run this program and then try attaching the switch to other side A data bits, and to control bit CA1. The program below configures the PIA, and waits for switch closure. It then places in accumulators A and B respectively, the control register contents both before and after the data register has been read.

Location	Hex	Instr.	Operand # , X	Comment
0 0 7 0	7 F	C L R	C R A	
0 0 7 1	E 8			
0 0 7 2	0 1			
0 0 7 3	7 F	C L R	D D R A	ALL LINES INPUTS
0 0 7 4	E 8			
0 0 7 5	0 0			
0 0 7 6	8 6	L D A	A # % 0 4	ACCESS DATA REGISTER
0 0 7 7	0 4			
0 0 7 8	b 7	S T A	A C R A	
0 0 7 9	E 8			
0 0 7 A	0 1			
0 0 7 B	b 6	L D A	A C R A	IS SWITCH CLOSED? WAITS
0 0 7 C	E 8			
0 0 7 D	0 1			
0 0 7 E	2 A	B P L	W A I T S	NO WAITS
0 0 7 F	F b			
0 0 8 0	F 6	L D A	B D D R A	CLEAR STATUS BIT
0 0 8 1	E 8			
0 0 8 2	0 0			
0 0 8 3	F 6	L D A	B C R A	GET NEW CR CONTENTS
0 0 8 4	E 8			
0 0 8 5	0 1			
0 0 8 6	3 F	S W I		

7. After you run the program, change it to debounce the switch by waiting for one millisecond. Did you notice the final contents of accumulators A and B, after the first run?

Location	Hex	Instr.	Operand	Comment
0 0 8 0	C E	L D X	# \$ 7 C	
0 0 8 1	0 0			
0 0 8 2	7 C			
0 0 8 3	0 9	D E X		DLY
0 0 8 4	2 6	B N E	D L Y	
0 0 8 5	F d			
0 0 8 6	F 6	L D A B	D D R A	
0 0 8 7	E 8			
0 0 8 8	0 0			
0 0 8 9	F 6	L D A B	C R A	
0 0 8 A	E 8			
0 0 8 B	0 1			
0 0 8 C	3 F	S W I		

8. Explain the difference in the final contents of accumulators A and B. What happens if you replace LDAB DDRA with STAB DDRA? How about ADDB DDRA, TST DDRA, CLR DDRA, or ROR DDRA? Identify the use of this feature and the consequence if the data register is not read. Change the program so that CA2 is used instead of CA1.

B. PIA Output Port

1. The following program makes an output port of PIA B, and the data and direction registers are stored in A and B respectively.

Location	Hex	Instr.	Operand # , X	Comment
0 0 7 0	7 F	C L R	C R B	
0 0 7 1	E 8			
0 0 7 2	0 3			
0 0 7 3	8 6	L D A A	# \$ F F	
0 0 7 4	F F			
0 0 7 5	b 7	S T A A	D D R B	ALL LINES OUTPUTS
0 0 7 6	E 8			
0 0 7 7	0 2			
0 0 7 8	F 6	L D A B	D D R B	SAVE DDR IN B
0 0 7 9	E 8			
0 0 7 A	0 2			
0 0 7 B	8 6	L D A A	# % 0 4	ACCESS DATA REGISTER
0 0 7 C	0 4			
0 0 7 D	b 7	S T A A	C R B	
0 0 7 E	E 8			
0 0 7 F	0 3			
0 0 8 0	b 6	L D A A	D D R B	SAVE DATA REGISTER IN A
0 0 8 1	E 8			
0 0 8 2	0 2			
0 0 8 3	3 F	S W I		

2. Run the program and identify the end values of the data and direction registers. Change LDAA #00000100 to LDAA #011000100 and change LDAA DDRB to LDAA CRB. Identify and explain the accumulator A final value.
3. Attach the cathode of an LED to data line PB7 and its anode to +5 volts. The following program configures the PIA and lights the LED.

Location	Hex	Instr.	Operand # , X	Comment
0 0 7 0	7 F	C L R	C R B	
0 0 7 1	E 8			
0 0 7 2	0 3			
0 0 7 3	8 6	L D A A	# \$ F F	ALL LINES OUTPUTS
0 0 7 4	F F			
0 0 7 5	b 7	S T A A	D D R B	
0 0 7 6	E 8			
0 0 7 7	0 2			
0 0 7 8	8 6	L D A A	# % 0 4	ACCESS DATA REGISTER
0 0 7 9	0 4			
0 0 7 A	b 7	S T A A	C R B	
0 0 7 B	E 8			
0 0 7 C	0 1			
0 0 7 D	7 F	C L R	D D R B	LIGHT THE LED
0 0 7 E	E 8			
0 0 7 F	0 2			
0 0 7 0	2 0	B R A	H E R E	AND WAIT HERE
0 0 7 1	F E			

4. Run the program. Change it so it turns the LED on for one second, and then so only one bit in the PIA data register is affected.

C. The Bidirectional Control Line

1. Control line 1 is always an input line, and control line 2 is input or output. Transitions on control line 1 usually indicate the presence of new data from an input device on the readiness of an output device to accept data. As an output line, control line 2 may indicate the presence of new output data, mark the completion of a transfer, or serve as a latched serial output.
2. Attach a switch to CB1 and a LED to CB2. The program below turns the LED on for 1/2 second, using CB2 as a serial output.

Location	Hex	Instr.	Operand # , X	Comment
0 0 7 0	7 F	C L R	C R B	
0 0 7 1	E 8			
0 0 7 2	0 3			
0 0 7 3	7 F	C L R	D D R B	DATA LINES INPUTS
0 0 7 4	E 8			
0 0 7 5	0 2			
0 0 7 6	8 6	L D A A	C R B	
0 0 7 7	3 4			
0 0 7 8	b 7	S T A A	C R B	
0 0 7 9	E 8			
0 0 7 A	0 3			
0 0 7 B	C E	L D X	# \$ F F F F	DELAY $\frac{1}{2}$ SECOND
0 0 7 C	F F			
0 0 7 D	F F			
0 0 7 E	0 9	D E X		
0 0 7 F	2 6	B N E	D E L Y	
0 0 8 0	F d			
0 0 8 1	8 6	L D A A	# % 3 C	
0 0 8 2	3 C			
0 0 8 3	b 7	S T A A	C R B	TURN LED OFF
0 0 8 4	E 8			
0 0 8 5	0 3			
0 0 8 6	3 F	S W I		

Identify the effect (if any) the switch has on the program. In this configuration the program controls the level and pulse length of the serial output.

3. Revise the program so that the LED is activated by key "7" and inactivated by key "0".

Note You may wait until key "D" is pressed with the instructions.

```

WAITO    LSR    STATUS    IS KEY "0" PRESSED?
          BCS    WAITO    NO, WAIT

WAITO    LSR    STATUS    74
          BCS    WAITO    80
          BCS    WAITO    04
          BCS    WAITO    25
          BCS    WAITO    FB

```

4. CB2 can also function as an acknowledge or computer ready signal. In this instance, it goes low after the CPU writes into the PIA and stays there until a transition on CB1. The following program is an example of this type CB2 signal:

Location	Hex	Instr.	Operand # , X	Comment
0 0 7 0	7 E	C L R	C R B	
0 0 7 1	E 8			
0 0 7 2	0 3			
0 0 7 3	7 F	C L R	D D R B	DATA LINES INPUTS
0 0 7 4	E 8			
0 0 7 5	0 2			
0 0 7 6	8 6	L D A A	# % 2 4	TURN LED ON
0 0 7 7	2 4			
0 0 7 8	b 7	S T A A	C R B	
0 0 7 9	E 8			
0 0 7 A	0 3			
0 0 7 B	7 F	C L R	D D R B	
0 0 7 C	E 8			
0 0 7 D	0 2			
0 0 7 E	2 0	B R A	H E R E	HERE
0 0 7 F	F E			

Identify the effect of opening and closing the switch on CB1, and the effect of the following data replacements: CLR DDRA with LDAA DDRA, STAA DDRB, ADDA DDRB, LSR DDRB, and TST DDRB. Indicate the usefulness of this capability. Notice that the PIA side A produces an equivalent signal in response to a read operation. Using side A, identify the instructions which activate the LED.

5. Another control line option produces a brief pulse which can indicate the presence of new data to the peripheral. You can effect this option by replacing LDAA #00100100 in the last program with LDAA #00101100. Identify the effect of this change.
6. This brief pulse on strobe can be verified by connecting CB2 to CB1. The following program will end with the old contents of the control register, before the strobe, in accumulator A, and the contents after the strobe in accumulator B.

Location	Hex	Instr.		Operand # , X	Comment
0 0 7 0	b 6	L D A	A	C R B	A = OLD CONTROL REGISTER
0 0 7 1	E 8				
0 0 7 2	0 3				
0 0 7 3	7 F	C L R		D D R B	PRODUCE STROBE
0 0 7 4	E 8				
0 0 7 5	0 2				
0 0 7 6	F 6	L D A	B	C R B	B = NEW CONTROL REGISTER
0 0 7 7	E 8				
0 0 7 8	0 3				
0 0 7 9	3 F	S W I			

7. Identify and rationalize the ending contents of accumulators A and B.

The output control line options are (control register bits):

bit-5 = 1 makes CB2 an output

bit-4 = 1 makes CB2 a latched serial output (level) with the value of bit-3

Bit-4 = 0 makes CB2 an active-low pulse which is either a long strobe deactivated by CB1 (bit-3 = 0) or a briefstrobe (bit-3 = 1)

Identify the circuit configurations which can be obtained from a single PIA under program control.

Chapter 13: Interrupts

Objective: Familiarization with uses of interrupts, the advantages of interrupts, the ASCI System X interrupt system and vectors, PIA interrupts, enabling and disabling interrupts, communication between main program and interrupts, and how to poll interrupts.

Student study requirements: Pages 65-66, and 168 through 211 of BASIC MICROPROCESSORS and THE 6800 by Bishop; pages 3-3 to 3-8, A28, and A61 of the M6800 PROGRAMMING MANUAL.

Discussion

I. General Information on Interrupts

With any discussion of handling interrupts, it is quite appropriate to review the subroutine because the primary difference between an interrupt service routine and a standard subroutine is the hardware initiation of the routine.

When a subroutine instruction is encountered, the microprocessor responds by saving the present program counter in the stack and installing the new beginning address in the PC register. When a return instruction within the subroutine is encountered, the processor retrieves the saved address from the stack and installs it in the program counter, then back to the main program.

In interrupt processing, this same sequence occurs with one difference: this action is not initiated by a preplanned instruction, but rather by external hardware setting the interrupt flip-flops on interrupt flags inside the microprocessor. During completion of each instruction cycle, the microprocessor checks this internal interrupt flag and, if it is set, it temporarily suspends the present operation in favor of the service routine being called for by the interrupting device. The difference between types of interrupts is determined by the methods used to present the starting address to the program counter.

There are basically two ways in which interrupts can be recognized by a microprocessor: by polled methods or by vectored methods. The term polled here should not be confused with the programmed control polling methods of handling I/Os. The polled I/O method usually means that the whole set of I/Os being operated are interrogated one at a time to determine if service is required or not. This same polling procedure may be initiated as the result of an interrupt signal. When used in this way, polling is usually called scanning to differentiate it from the standard polled I/O system.

The scanned on polled method of interrupt handling is a software approach for generating the address vector for the PC register; whereas, the vectored method implies that the address vector or the address vector memory location is automatically installed in the PC register. The term vector in this context is, in many cases, misused. A vector is a pointer address. That is, it is the entry address of a software routine.

II. New Instructions

- A.** CLI (Clear Interrupt Mask) clears the interrupt mask bit in the CCR. This will allow the MPU to service an interrupt request from an external device if signaled by a high on IRQ line. (See page A28 of the M6800 PROGRAMMING MANUAL.) CLI operates only in the inherent mode, so an OE instruction code will set bit-4 of CCR to a zero.
- B.** SEI (Set Interrupt Mask) sets the interrupt mask bit in the MPU CCR so the MPU is inhibited from servicing an interrupt request and will continue with program. (See page A61 of the M6800 PROGRAMMING MANUAL.) SEI operates in the inherent mode, so an OF instruction code will set bit-4 of the CCR to a one.

III. Laboratory Exercise

A. Interrupts in the System X

- 1.** The CPU is provided direct serial inputs via interrupts. Using an interrupt, an external device can directly notify the CPU that it has data, is ready to receive data, or has some other requirement. The program is relieved of checking the status bit and precautions are not necessary to avoid missing an event. Rather, the computer continues as usual, or waits for an external event. System X responses to an interrupt are as follows:
 - All register contents in the stack are saved.
 - The contents of memory locations 0031 and 0032 are placed in the program counter. This 16-bit address is the IRQQUAD pointer.

The address of the interrupt service routine is to be placed in IRQUAD.

2. Enabling or disabling the interrupt system:

- CLI (0E) enables interrupts (i.e., clears the interrupt disable bit).
- SEI (0F) disables interrupts (i.e., sets the interrupt disable bit).

Note: On accepting an interrupt, or on RESET, the System X automatically disables interrupts.

3. Each PIA has an interrupt enable bit which is bit-0 of the control register. To permit an interrupt from the PIA, it must be 1. Clearing of the interrupt bit (control register bit-7) is accomplished by reading the PIA data.

B. Basic Interrupt Using PIA

- 1.** As switch attached to line CA1 will produce a simple interrupt. The program below waits for you to close the switch and then returns to the monitor. Remember, enable both the overall interrupt in the CPU with the CLI instruction and the PIA interrupt. Clearing the interrupt may be accomplished by reading the data register of the PIA (after debouncing).

Location	Hex	Instr.	Operand #, X	Comment
0 0 7 0	0 F	S E I		DISABLE INTERRUPT
0 0 7 1	C E	L D X	# \$ 0 0 A 0	STORE INTERRUPT SERVICE ADDRESS
0 0 7 2	0 0			
0 0 7 3	A 0			
0 0 7 4	D F	S T X	I R Q A D	

0 0, 7, 5	3, 1				
0 0, 7, 5	8, 6	L D A	A	# % 0 5	ENABLE PIA INTERRUPT
0 0, 7, 7	0 5				
0 0, 7, 8	b 7	S T A	A	C R A	
0 0, 7, 9	E 8				
0 0, 7, A	0 1				
0 0, 7, B	0 E	C L I			ENABLE INTERRUPT
0 0, 7, C	2 0	B R A		W A I T	
0 0, 7, D	F E				
	E				
	F				
0 0, A, 0	C E L D X			# \$ 0 0 3 E	DEBOUNCE SWITCH
0 0, A, 1	0 0				
0 0, A, 2	3 E				
0 0, A, 3	0 9	D E X			
0 0, A, 4	2 6	B N E		D E L A Y	
0 0, A, 5	F d				
0 0, A, 6	b 6	L D A	A	D D R A	CLEAR INTERRUPT
0 0, A, 7	E 8				
0 0, A, 8	0 0				
0 0, A, 9	3 F	S W I			

2. What are the final contents of the stack? The external interrupt and an SWI instruction are alike in that they cause the CPU to save all the stack registers. Define the register A value at the end of the interrupt service routine. The stack still contains the original value of A in the main program. Define where.
3. Define the interrupt flag value at the end of the interrupt service routine. Remember that interrupts are automatically disabled by the CPU when one is accepted. Also, the stack still contains the original value of the interrupt flag.

4. Adjust the program to use control line CB1 (connector pin 18) rather than CA1; and then use CB2 (connector pin V).

Note: Control register bit-3 must be set to 1 to enable the interrupt on control line 2.

C. Communication Between Main Program and Interrupt

1. When the System X accepts an interrupt, it saves all the register contents and, after servicing the interrupt, restores them all via the RTI instruction. Therefore, the memory is used as the link between the main program and the interrupt service routine. The program below uses a flag in memory location b0 to determine if the interrupt from CA1 has occurred.

- b0 = before interrupt
- b0 = 1 after interrupt

Location	Hex	Instr.	Operand # , X	Comment
0 0 7 0	0 F	S E I		DISABLE INTERRUPT
0 0 7 1	C E	L D X	# \$ A 0	STORE INTERRUPT SERVICE ADD
0 0 7 2	0 0			
0 0 7 3	A 0			
0 0 7 4	d F	S T X	I R Q A D	
0 0 7 5	3 1			
0 0 7 6	8 6	L D A A	# % 0 5	ENABLE PIA INTERRUPT
0 0 7 7	0 5			
0 0 7 8	b 7	S T A	A D D R A	
0 0 7 9	E 8			
0 0 7 A	0 1			
0 0 7 B	7 F	C L R	0 0 6 0	INTERRUPT FLAG = 0
0 0 7 C	0 0			

0 0 7 D	b 0			
0 0 7 E	0 E	C L I		ENABLE INTERRUPT
0 0 7 F	7 d	T S T	\$ 6 0	HAS INTERRUPT OCCURRED WAIT
0 0 8 0	0 0			
0 0 8 1	b 0			
0 0 8 2	2 7	B N E	W A I T	NO WAIT
0 0 8 3	F b			
0 0 8 4	3 F	S W I		

INTERRUPT SERVICE ROUTINE

Location	Hex	Instr.	Operand # , X	Comment
0 0 A 0	C E	L D X	# \$ 7 C	WAIT 1 MILLISECOND
0 0 A 1	0 0			
0 0 A 2	7 C			
0 0 A 3	0 9	D E X		DELAY
0 0 A 4	2 6	B N E	D E L A Y	
0 0 A 5	F d			
0 0 A 6	b 6	L D A A	D D R A	CLEAR INTERRUPT
0 0 A 7	E 8			
0 0 A 8	0 0			
0 0 A 9	7 C	I N C	\$ b 0	MARKER = 1
0 0 A A	0 0			
0 0 A B	b 0			
0 0 A C	3 b	R T I		
	D			
	E			

- Try running the program using accumulator B instead of memory location b0.

Note Program reorganization is unnecessary — put NOP (01) in the unused spots, i.e., replace CLR \$b0 with CLRB, NOP, NOP.

3. Can you define the problem? (The value of register B in the stack has not been changed.) Try the following rather than INCB:

TSX		USE STACK POINTER AS DATA
INX		POINTER AND INCREMENT B IN STACK
INC	X	
A9	TSX	30
AA	INX	08
AB	INC	X 6C
AC		00
AD	RTI	3B

Did this work? Define why. Suppose you want the interrupt disabled on returning. How would you set the stack interrupt flag?

D. Interrupt Usage in the System X

1. The program does not have to examine the status bit with usage of the interrupt. Rather, the System X is notified by the status bit that it is active. Thus, the computer continues valuable work until interrupted. The following program simply counts using three memory locations (b0, b1 and b2). How high will it count before you can interrupt it?

Location	Hex	Instr.	Operand # , X	Comment
0 0 7 0	0 F	S E I		CLEAR INTERRUPT
0 0 7 1	C E	L D X	# \$ A 0	STORE INTERRUPT SERVICE ADD
0 0 7 2	0 0			
0 0 7 3	A 0			
0 0 7 4	d F	S T X	I R Q A D	
0 0 7 5	3 1			
0 0 7 6	8 6	L D A A	# % 0 5	ENABLE PIA INTERRUPT
0 0 7 7	0 5			
0 0 7 8	b 7	S T A A	D D R A	ALL LOCATIONS = 0
0 0 7 9	E 8			

0 0 7 A	0 1				
0 0 7 B	7 F				
0 0 7 C	0 0				
0 0 7 D	b 0				
0 0 7 E	7 F				
0 0 7 F	0 0				
0 0 8 0	b 1				
0 0 8 1	7 F				
0 0 8 2	0 0				
0 0 8 3	b 2				
0 0 8 4	0 E	C L I			ENABLE INTERRUPT
0 0 8 5	7 C				AND COUNT CNT
0 0 8 6	0 0				
0 0 8 7	b 2				
0 0 8 8	2 6	B N E	C N T		
0 0 8 9	F b				
0 0 8 A	7 C				
0 0 8 B	0 0				
0 0 8 C	b 1				
0 0 8 D	2 6	B N E	C N T		
0 0 8 E	F 6				
0 0 8 F	7 C				
0 0 9 0	0 0				
0 0 9 1	b 0				
0 0 9 2	2 0	B R A	C N T		
0 0 9 3	F 1				

The stack can be cleared by RESET. Try the program several times. Using the interrupt, the response is immediate, and valuable time is not wasted looking for the switch input.

2. Adjust the program to initially disable the interrupt, but to enable it by pressing key "F". Check for key "F" during each cycle. Reopen the

switch after having it closed while the interrupt is disabled. What happens when you press key "F"? Does it matter if the initialization disables the PIA interrupt or the entire system? Adjust the program to include in each cycle, an LDAA DDRA instruction. Does the PIA remember the interrupt? Unless the program clears it, an interrupt that has not been serviced will continue to be active.

E. Polling Interrupts in the System X

1. The CPU has to check the PIA status bits when more than one source for an interrupt exists, However, here the CPU knows one bit is active. Attach switches to both CA1 and CA2. The program below waits for the interrupt and than displays either 1 or 2 on the LEDs.

Location	Hex	Instr.	Operand # , X	Comment
0 0 7 0	0 F	S E I		DISABLE INTERRUPT
0 0 7 1	C E	L D X	# \$ A 0	STORE SERVICE ADDRESS
0 0 7 2	0 0			
0 0 7 3	A 0			
0 0 7 4	D F	S T X	I R Q A D	
0 0 7 5	3 1			
0 0 7 6	8 b	L D A A	# % 0 D	ENABLE PIA INTERRUPT
0 0 7 7	0 d			
0 0 7 8	b 7	S T A A	S T A T A	
0 0 7 9	E 8			
0 0 7 A	0 1			
0 0 7 B	0 E	C L I		ENABLE INTERRUPT
0 0 7 C	2 0	B R A	W A I T	AND WAIT WAIT
0 0 7 D	F E			

Location	Hex	Instr.	Operand # , X	Comment
0 0 A 0	C 6	L D A B	# 6 0	CODE TO DISPLAY 1
0 0 A 1	6 0			
0 0 A 2	b 6	L D A A	S T A T A	CLEAR INTERRUPT FROM LINE 1
0 0 A 3	E 8			
0 0 A 4	0 1			
0 0 A 5	2 b	B M I	D S P L Y	YES DISPLAY 1
0 0 A 6	0 5			
0 0 A 7	4 8	A S L A		CLEAR INTERRUPT FROM LINE 2
0 0 A 8	2 A	B P L	L A S T	NO, WAIT
0 0 A 9	0 E			
0 0 A A	C 6	L D A B	# \$ d A	YES, GET CODE TO DISPLAY 2
0 0 A B	d A			
0 0 A C	b 6	L D A A	D A T A A	CLEAR INTERRUPT DSPLY
0 0 A D	E 8			
0 0 A E	0 0			
0 0 A F	8 6	L D A A	# \$ F F	TURN ON DISPLAYS
0 0 B 0	F F			
0 0 B 1	b 7	S T A A	L I G H T S	
0 0 B 2	E 4			
0 0 B 3	0 0			
0 0 B 4	F 7	S T A B	G L O W	DISPLAY 1 or 2
0 0 B 5	E 4			
0 0 B 6	0 2			
0 0 B 7	2 0	B R A	L A S T	WAIT LAST
0 0 B 8	F E			

- After running the program, adjust the main program to wait for key "F" to be pressed before enabling the interrupts. If you close one or both of the switches prior to pressing key "F", what happens? The interrupt which has "higher priority" is the one that takes precedence over the other. Change the priority. Interrupt priority is established by the order in which interrupt flag bits are examined. A polling interrupt system is one which uses flag bits to identify the interrupt source.

GLOSSARY OF COMMONLY USED MICROPROCESSOR TERMS

ABSOLUTE ADDRESSING - SEE DIRECT ADDRESSING

ABSOLUTE INDEXED ADDRESSING - The effective address is formed by adding the index register (X or Y) to the second and third byte of the instruction.

ACCUMULATOR - A register that holds one of the operands and the result of arithmetic and logic operations that are performed by the central processing unit. Also commonly used to hold data transferred to or from I/O devices.

ACCUMULATOR ADDRESSING - One byte instruction operating on the accumulator.

ACIA - Is an Asynchronous Communications Interface Adapter. This is an NMOS LSI device produced by Motorola for interfacing Serial ASCII devices to a micro-processor system.

ADDRESS - A number that designates a memory or I/O location.

ADDRESS BUS - A multiple-bit output Bus for transmitting an address from the CPU to the rest of the system.

ALGORITHM - The sequence of operations which defines the solution to a problem.

ALPHANUMERIC - Pertaining to a character set that contains both letters and numerals and usually other characters.

ALU (ARITHMETIC/LOGIC UNIT) - The unit of a computing system that performs arithmetic and logic operations.

ASCII CODE - The American Standard Code for Information Interchange. A seven-bit character code without the parity bit, or an eight-bit character code with the parity bit.

ASSEMBLER - A program that translates symbolic operation codes into machine language, symbolic addresses to memory addresses and assigns values to all program symbols. It translates source programs to object programs.

ASSEMBLY DIRECTIVE - A mnemonic that modifies the assembler operation but does not produce an object code (e.g., a pseudo instruction).

ASSEMBLY LANGUAGE - A collection of symbolic labels, mnemonics, and data which are to be translated into binary machine codes by the assembler.

ASYNCHRONOUS - Not occurring at the same time, or not exhibiting a constant repetition rate; irregular.

BASE - "SEE RADIX".

BCD - Binary Code Decimal. A means by which decimal numbers are represented as binary values, where integers in the range 0-9 are represented by the four-bit binary codes from 0000-1001.

BIDIRECTIONAL DATA BUS - A data bus in which digital information can be transferred in either direction.

BINARY - The base two number systems. All numbers are expressed as powers of two. As a consequence, only two symbols (0 & 1) are required to represent any number.

BIT - The smallest unit of information which can be represented. A bit may be in one of two states, represented by the binary digits 0 and 1.

BLOCK DIAGRAM - A diagram in which the essential units of any system are drawn in the form of blocks, and their relationship to each other is indicated by appropriately connected lines.

BRANCH INSTRUCTION - An instruction that causes a program jump to a specified address and execution of the instruction at that address. During the execution of the branch instruction, the central processor replaces the contents of the program counter with the specified address.

BREAKPOINT - Pertaining to a type of instruction, instruction digit, or other condition used to interrupt or stop a computer at a particular place in a program. A place in a program where such an interruption occurs or can be made to occur.

BUFFER - A noninverting digital circuit element that may be used to handle a large fan-out or to invert input and output levels.

A storage device used to compensate for a difference in rate of flow of data, or time of occurrence of events, when transmitting data from one device to another.

BYTE - A sequence of eight adjacent binary digits operates upon as a unit.

CALL - A special type of jump in which the central processor is logically required to "remember" the contents of the program counter at the time that the jump occurs. This allows the processor later to resume execution of the main program, when it is finished with the last instruction of the subroutine.

CASCADE - An arrangement of two or more similar circuits in which the output of one circuit provides the input of the next.

CLOCK - A device or a part of a device that generates all the timing pulses for the coordination of a digital system. System clocks usually generate two or more clock phases. Each phase is a separate square wave pulse train output.

CODING - The process of preparing a program from the flow chart defining an algorithm.

COMPILER - A language translator which converts individual source statements into multiple machine instructions. A compiler translates the entire program before it is executed.

COMPLEMENT - Reverse all binary bit values (ones become zeros, zeros become ones).

CONDITIONAL - In a computer, subject to the result of a comparison made during computation.

CONDITIONAL BREAKPOINT INSTRUCTION - A conditional jump instruction that causes a computer to stop if a specified switch is set. The routine then may be allowed to proceed as coded, or a jump may be forced.

CONDITIONAL JUMP - Also called conditional transfer of control. An instruction to a computer which will cause the proper one of two (or more) addresses to be used in obtaining the next instruction, depending on some property of one or more numerical expressions or other conditions.

CONTACT BOUNCE - The uncontrolled making and breaking of a contact when the switch or relay contacts are closed. An important problem in digital circuits, where bounces can act as clock pulses.

CPU (CENTRAL PROCESSING UNIT) - The unit of a computing system that controls the interpretation and execution of instructions; includes the ALU.

DATA BUS - A multi-line, parallel path over which digital data is transferred, from any of several destinations. Only one transfer of information can take place at any one time. While such transfer is taking place, all other sources that are tied to the bus must be disabled.

DEBUG - Detect, locate, and correct problems in a program or hardware.

DEBOUNCED - Refers to a switch or relay that no longer exhibits contact bounce.

DECODER/DRIVER - A code conversion device that can also have sufficient voltage or current output to drive an external device such as a display or a lamp monitor.

DEMULTIPLEXER - A digital device that directs information from a single input to one of several outputs. Information for output-channel selection usually is presented to the device in binary weighted form and is decoded internally. The device also acts as a single-pole multiposition switch that passes digital information in a direction opposite to that of a multiplexer.

DESTINATION - Register, memory location or I/O device which can be used to receive data during instruction execution.

DEVICE SELECT PULSE - A software-generated positive or negative clock pulse from a computer that is used to strobe the operation of one or more I/O devices, including individual integrated circuit chips.

DIRECT ADDRESSING - The second and third byte of the instruction contain the address of operand to be used.

DMA (DIRECT MEMORY ACCESS) - Suspension of processor operation to allow peripheral units external to the CPU to exercise control of memory for both READ and WRITE without altering the internal state of the processor.

DYNAMIC RAM - A random access memory that uses a capacitive element for storing a data bit. They require REFRESH.

EBCDIC - The Extended Binary Coded Decimal Interchange Code, a digital code primarily used by IBM. It closely resembles the half-ASCII code.

EDGE - The transition from logic 0 to logic 1, or from logic 1 to logic 0, in a clock pulse.

EDITOR - A program used for preparing and modifying a source program or other file by addition, deletion or change.

EFFECTIVE ADDRESS - The actual address of the desired location in memory, usually derived by some form of calculation.

EXPANSION - The process of inserting a sequence of operations represented by a macro name when the macro name is referenced in a program.

FALL TIME - The time required for an output voltage of a digital circuit to change from a logic 1 to a logic 0 state.

FAN-OUT - The number of parallel loads within a given logic family that can be driven from one output mode of a logic circuit.

FETCH - One of the two functional parts of an instruction cycle. The collective actions of acquiring a memory address, and then an instruction or data byte from memory.

FIELD - An area of an instruction mnemonic.

FILE - A collection of data records treated as a single unit.

FIFO (FIRST IN, FIRST OUT) - The term applies to the sequence of entering data into and retrieving data from data storage. The first data entered is the first data obtainable with FIFO.

FLAG - A status bit which indicates that a certain condition has arisen during the course of arithmetic or logical manipulations or data transmission between a pair of digital electronic devices. Some flags may be tested and thus be used for determining subsequent actions.

FLAG REGISTER - A register consisting of the flag flip-flops.

FLOW CHART - A symbolic representation of the algorithm required to solve a problem.

FREQUENCY - The number of recurrences of a periodic phenomenon in a unit of time. Electrical frequency is specified as so many cycles per second, or Hertz.

FULL DUPLEX - A data transmission mode which provides simultaneous and independent transmission and reception.

HALF-ASCII - A 64-character ASCII code that contains the code words for numeric digits, alphabetic characters, and symbols but not keyboard operations.

HALF DUPLEX - A data transmission mode which provides both transmission and reception but not simultaneously.

HANDSHAKE - Interactive communication between two system components, such as between the CPU and a peripheral; often required to prevent loss of data.

HARDWARE - Physical equipment mechanical, electrical, or electronic devices.

HEXADECIMAL - A number system based upon the radix-16, in which the decimal numbers 0 through 9 and the letters A through F represent the sixteen distinct states in the code.

HIGH ADDRESS BYTE - The eight most significant bits in the 16-bit memory address word. Abbreviated H or HI.

IC (INTEGRATED CIRCUIT) - (1) A combination of interconnected circuit elements inseparably associated on or within a continuous substrate. (2) Any electronic device in which both active and passive elements are contained in a single package. In digital electronics, the term chiefly applies to circuits containing semiconductor elements.

IMMEDIATE ADDRESSING - The Operand is the second byte of the instruction, rather than its address.

IMPLIED ADDRESSING - A one-byte instruction that stipulates an operation internal to the processor. DOES NOT require any additional operand.

INCREMENT - To increase the value of a binary word. Typically, to increase the value by 1.

INDEXED ADDRESS - An indexed address is a memory address formed by adding immediate data included with the instruction to the contents of some register or memory location.

INDEXED INDIRECT ADDRESSING - The second byte of the instruction is added to the contents of the "X" index register, discarding the carry, to form a zero-page effective address.

INDIRECT ABSOLUTE ADDRESSING - The second and third bytes of the instruction contain the address for the first of two bytes in memory that contain the effective address.

- INDIRECT INDEXED ADDRESSING** - The second byte of this instruction is a zero-page address. The contents of this zero-page address are added to the "Y" index register to form the lower 8 bits of the effective address. Then the carry (if any) is added to the contents of the next zero-page address to form the higher 8 bits of the effective address.
- INDIRECT ADDRESS** - An address used with an instruction that indicates a memory location or a register that in turn contains the actual address of an operand. The indirect address may be included with the instruction, contained in a register (register indirect address) or contained in a memory location (memory directed indirect address).
- INTERFACING** - The joining of members of a group (such as people, instruments, etc.) in such a way that they are able to function in a compatible and coordinated fashion.
- INSTRUCTION** - A statement that specifies an operation and the values or locations of its operands.
- INSTRUCTION CODE** - A unique binary number that encodes an operation that a computer can perform.
- INSTRUCTION CYCLE** - A successive group of machine cycles, as few as one or as many as seven, which together perform a single microprocessor instruction within the microprocessor chip.
- INSTRUCTION DECODER** - A decoder within a CPU that decodes the instruction code into a series of actions that the computer performs.
- INSTRUCTION REGISTER** - The register that contains the instruction code.
- INTERPRETER** - A language translator which converts individual source statements into multiple machine instructions by translating and executing each statement as it is encountered. Can not be used to generate object code.
- INTERRUPT** - In a computer, a break in the normal flow of a system or routine such that the flow can be resumed from that point at a later time. The source of the interrupt may be internal or external.
- I/O DEVICE** - Input/Output device - any digital device, including a single integrated circuit chip, that transmits data on strobe pulses to a computer or receives data or strobe pulses from a computer.
- JUMP** - (1) To cause the next instruction to be selected from a specified storage location in a computer. (2) A deviation from the normal sequence of execution of instructions in a computer.
- LABEL** - One or more characters that serve to define an item of data or the location of an instruction or subroutine. A character is one symbol of a set of elementary symbols, such as those corresponding to typewriter keys.
- LATCH** - A simple logic storage element. A feedback loop used in a symmetrical digital circuit, such as a flip-flop, to retain a state.
- LEADING EDGE** - The transition of a pulse that occurs first.
- LED (LIGHT-EMITTING DIODE)** - A pn junction that emits light when biased in the forward direction.

- LEVEL-TRIGGERED** - The state of the clock input, being either logic 0 or logic 1 carries out a transfer of information or completes an action.
- LIFO (LAST IN, FIRST OUT)** - The latest data entered is the first data obtainable from a LIFO stack or memory section.
- LSB (LEAST SIGNIFICANT BIT)** - The digit with the lowest weighting in a binary number.
- LISTING** - An assembler output containing a listing of program mnemonics, the machine code produced, and diagnostics, if any.
- LOGIC** - (1) The science dealing with the basic principles and applications of truth tables, switching, gating, etc. (2) See Logical Design. (3) Also called symbolic logic. A mathematical approach to the solution of complex situations by the use of symbols to define basic concepts. The three basic logic symbols are AND, OR, and NOT. When used in Boolean algebra, these symbols are somewhat analogous to addition and multiplication. (4) In computers and information-processing networks, the systematic method that governs the operations performed on information, usually with each step influencing the one that follows. (5) The systematic plan that defines the interactions of signals in the design of a system for automatic data processing.
- LOGICAL DECISION** - The ability of a computer to make a choice between two alternatives; basically, the ability to answer yes or no to certain fundamental questions concerning equality and relative magnitude.
- LOGICAL DESIGN** - The synthesizing of a network of logical elements to perform a specified function. In digital electronics, these logical elements are digital electronic devices, such as gates, flip-flops, decoders, counters, etc.
- LOGICAL ELEMENT** - In a computer or data-processing system, the smallest building blocks which operators can represent in an appropriate system of symbolic logic. Typical logical elements are the AND gate and the "flip-flop".
- LOOP** - A sequence of instructions that is repeated until a conditional exit situation is met.
- LOW ADDRESS BYTE** - The eight least significant bits in the 16-bit memory address word. Abbreviated L or LO.
- LSI (LARGE SCALE INTEGRATION)** - Integrated circuits that perform complex functions. Such chips usually contain 100 to 2,000 gates.
- MACHINE CODE** - A binary code that a computer decodes to execute a specific function.
- MACHINE CYCLE** - A subdivision of an instruction cycle during which time a related group of actions occur within the microprocessor chip. In the 8080 microprocessor, there exist nine different machine cycles. All instructions are combinations of one or more of these machine cycles.
- MACRO ASSEMBLER** - An assembler routine capable of assembling programs which contain and reference macro instructions.
- MACRO INSTRUCTION** - A symbol that is used to represent a specified sequence of source instructions.
- MAGNETIC CORE** - A type of computer storage which employs a core of magnetic material with wires threaded through it. The core can be magnetized to represent a binary 1 or 0.

MAGNETIC DRUM - A storage device consisting of a rapidly rotating cylinder, the surface of which can be easily magnetized and which will retain the data. Information is stored in the form of magnetized spots (or no spots) on the drum surface.

MAGNETIC DISC - A flat circular plate with a magnetic surface on which data can be stored by selective magnetization of portions of the flat surface.

MAGNETIC TAPE - A storage system based on the use of magnetic spots (bits) on metal or coated-plastic tape. The spots are arranged so that the desired code is read out as the tape travels past the read-write head.

MASKING - A process that uses a bit pattern to select bits from a data byte for use in a subsequent operation.

MEMORY - Any device that can store logic 1 and logic 0 bits in such a manner that a single bit or group of bits can be accessed and retrieved.

MEMORY ADDRESS - A 16-bit binary number that specifies the precise memory location of a memory word among the 65,536 different possible memory locations.

MEMORY CELL - A single storage element of memory, capable of storing one bit of digital information.

MICROCOMPUTER - A computer system based on a microprocessor and contains all the memory and interface hardware necessary to perform calculations and specified information transformations.

MICROPROCESSOR - A central processing unit fabricated as one integrated circuit.

MICROPROGRAM - A computer program written in the most basic instructions or subcommands that can be executed by the computer. Frequently, it is stored in a read-only memory.

MNEMONIC - Symbols representing machine instructions designed to allow easy identification of the functions represented.

MODULO - The modulo of a counter is simply n , the number of distinct states the counter goes through before repeating. A four-bit binary counter has a modulo of 16; a decade counter has a modulo of 10; and a divide-by-7 counter has a modulo of 7. In a variable modulo counter, n can be any value within a range of values.

MONITOR - Software or hardware that observes, supervises, controls, or verifies system operation.

MONOSTABLE MULTI-VIBRATOR - Also called one-shot multi-vibrator, single-shot multi-vibrator, or start-stop multi-vibrator. A circuit having only one stable state, from which it can be triggered to change the state, but only for a predetermined interval, after which it returns to the original state.

MSI (MEDIUM SCALE INTEGRATION) - Integrated circuits that perform simple, self-contained logic systems, such as counters and flip-flops.

MSB (MOST SIGNIFICANT) - The digit with the highest weighting in a binary number.

MULTIPLEXER - A digital device that can select one of a number of inputs and pass the logic level of that input on to the output. Information for input-channel selection usually is presented to the device in binary weighted form and decoded internally. The device acts as a single-pole multiposition switch that passes digital information in one direction only.

NEGATIVE EDGE - The transition from logic 1 to logic 0 in a clock pulse.

NEGATIVE-EDGE TRIGGERED - Transfer of information occurs on the negative edge of the clock pulse.

NEGATIVE LOGIC - A form of logic in which the more positive voltage level represents logic 0 and the more negative level represents logic 1.

NESTING - A sequential calling of subroutines without returning to the main program.

NIBBLE - A sequence of four adjacent bits, or half a byte, is a nibble. A hexadecimal or BCD digit can be represented in a nibble.

NON-OVERLAPPING TWO-PHASE CLOCK - A two-phase clock in which the clock pulses of the individual phases do not overlap.

NON-VOLATILE MEMORY - A semiconductor memory device in which the stored digital data is not lost when the power is removed.

OCTAL - A number system based upon the radix 8, in which the decimal numbers 0 through 7 represent the eight distinct states.

ONE-BYTE-INSTRUCTION - An instruction that consists of eight contiguous bits occupying one successive location.

OPEN-COLLECTOR OUTPUT - An output from an integrated circuit device in which the final "pull-up" resistor in the output transistor for the device is missing and must be provided by the user before the circuit is completed.

OPERAND - Data which is, or will be, operated upon by an arithmetic/logic instruction; usually identified by the address portion of an instruction, explicitly or implicitly.

OPERATION - Moving or manipulating data in the CPU or between the CPU and peripherals.

PAGE - A page consists of all the locations that can be addressed by 8-bits (a total of 256 locations) starting at 0 and going through 255. The address within a page is determined by the lower 8-bits of the address and the page number (0 through 255) is determined by the higher 8-bits of a 16-bit address.

PARITY - A method of checking the accuracy of binary numbers. If even parity is used, the sum of all the 1's in a number and its corresponding parity bit is always even. If odd parity is used, the sum of all the 1's and the parity bit is always odd.

PARTITIONING - The process of assigning specified portions of a system responsibility for performing specified functions.

PC - See "PROGRAM COUNTER"

PIA - PERIPHERAL INTERFACE ADAPTOR (MOS Technology's MPS 6520)

PERIPHERAL - A device or subsystem external to the CPU that provides additional system capabilities.

POLLING - Periodic interrogation of each of the devices that share a communications line to determine whether it requires servicing. The multiplexer or control station sends a poll that has the effect of asking the selected device, "Do you have anything to transmit?"

POP - Retrieving data from a stack.

PORT - A device or network through which data may be transferred or where device or network variables may be observed or measured.

POSITIVE EDGE - The transition from logic 0 to logic 1 in a clock pulse.

POSITIVE-EDGE TRIGGERED - Transfer of information occurs on the positive edge of the clock pulse.

POSITIVE LOGIC - A form of logic in which the more positive voltage level represents logic 1 and the more negative level represents logic 0.

PRIORITY - A preferential rating. Pertains to operations that are given preference over other system operations.

PROCESSOR - Shorthand word for microprocessor

PROGRAM - A group of instructions which causes the computer to perform a specified function.

PROGRAM COUNTER - A register containing the address of the next instruction to be executed. It is automatically incremented each time program instructions are executed.

PROGRAM LABEL - A symbol which is used to represent a memory address.

PROM (PROGRAMMABLE READ-ONLY MEMORY) - A read-only memory that is field programmable by the user.

PROPAGATION DELAY - A measure of the time required for a logic signal to travel through a logic device or a series of logic devices. It occurs as the result of four types of circuit delays - storage, rise, fall, and turn-on-delay - and is the time between when the input signal crosses the threshold - voltage point and when the responding voltage at the output crosses the same voltage point.

PSEUDO-INSTRUCTION - A mnemonic that modifies the assembler operation but does not produce an object code.

PULL-UP RESISTOR - A resistor connected to the positive supply voltage to the output collector of open-collector logic. Also used occasionally with mechanical switches to insure the voltage of one or more switch positions.

PULSE WIDTH - Also called pulse length. The time interval between the points at which the instantaneous value on the leading and trailing edges bears a specified relationship to the peak pulse amplitude.

PUSH - Putting data into a stack.

RADIX - Also called the base. The total number of distinct marks or symbols used in a numbering system. For example, since the decimal numbering system uses ten symbols, the radix is 10. In the binary numbering system, the radix is 2, because there are only two marks or symbols (0 and 1). In the octal numbering system, the radix is 8, and in the hexadecimal numbering system, the radix is 16.

RAM (RANDOM ACCESS MEMORY) - A semiconductor memory into which logic 0 and logic 1 states can be written (stored) and then read out again (retrieved).

READ - In semiconductors: To transmit data from a semiconductor memory to some other digital electronic device. The term, "read", also applies to computers and other types of memory devices.

- REFRESH** - The process by which dynamic RAM cells recharge the capacitive node to maintain the stored information. The charged nodes discharge due to leakage currents and without refresh, the stored data would be lost. This process must reoccur every so many microseconds. During refresh, the RAM cannot be accessed.
- REFRESH LOGIC** - The logic required to generate all the refresh signals and timing.
- REGISTER** - A hardware element used to temporarily store data.
- RELATIVE ADDRESS** - A relative address is a memory address formed by adding the immediate data included with the instruction to the contents of the program counter or some other register.
- RESET** - A computer system input that initializes and sets up certain registers in the CPU and throughout the computer system. One of the initializations, is to load a specific address into the Program Counter. The two bytes of information in that and the succeeding address is the starting address for the system program (for the MOS TECHNOLOGY processors).
- RETURN** - A special type of jump in which the central processor resumes execution of the main program at the contents of the program counter at the time that the jump occurred.
- RIPPLE COUNTER** - A binary counting system in which flip-flops are connected in series.
- RISE TIME** - The time required for an output voltage of a digital circuit to change from a logic 0 to a logic 1 state.
- ROM (READ-ONLY MEMORY)** - A semiconductor memory from which digital data can be repeatedly read out, but cannot be written into, as is the case for a RAM.
- ROUTINE** - A group of instructions that causes the computer to perform a specified function, e.g. a program.
- SCRATCH PAD** - The term applies to memory that is used temporarily by the CPU to store intermediate results.
- SEVEN-SEGMENT DISPLAY** - An electronic display that contains seven lines or segments spatially arranged in such a manner that the digits 0 through 9 can be represented through the selective lighting of certain segments to form the digit.
- SEMICONDUCTOR MEMORY** - A digital electronic memory device in which 1's and 0's are stored, that is a product of semiconductor manufacturing.
- SHIFT REGISTER** - A digital storage circuit in which information is shifted from one flip-flop of a chain to the adjacent flip-flop upon application for each clock pulse. Data may be shifted several places to the right or left, depending on additional gating and the number of clock pulses applied to the register. Depending on the number of positions shifted, the rightmost characters are lost in a right shift, and the leftmost characters are lost in a left shift.
- SIMULATOR** - A program which represents the functioning of one computer system utilizing another computer system.
- SOFTWARE** - The means by which any defined procedure is specified for computer execution.
- SOURCE** - Register, memory location or I/O device which can be used to supply data for use by an instruction.
- SOURCE PROGRAM** - A group of statements conforming to the syntax requirements of a language processor.

SPLIT DATA BUS - Is two data buses, one for incoming communications and one for outgoing communications. An 8-bit data bus in split data bus system takes 16 lines.

STACK - A specified section of sequential memory locations used as a LIFO (Last In, First Out) file. The last element entered is the first one available for output. A stack is used to store program data, subroutine return addresses, processor status, etc.

STACK POINTER (SP) - A register which contains the address of the system read/write memory used as a stack. It is automatically incremented or decremented as instructions perform operations with the stack.

STATEMENT - An instruction in source language.

STATIC RAM - A random access memory that uses a flip-flop for storing a binary data bit. Does not require refresh.

STRING - A series of values.

SUBROUTINE - A routine that causes the execution of a specified function and which also provides for transfer of control back to the calling routine upon completion of the function.

SYMBOL - Any character string used to represent a label, mnemonic, or data constant.

SYMBOLIC ADDRESS - Also called floating address. In digital computer programming, a label chosen in a routine to identify a particular word, function, or other information that is independent of the location of the information within the routine.

SYMBOLIC CODE - A code by which programs are expressed in source language; that is, storage locations and machine operations are referred to by symbolic names and addresses that do not depend upon their hardware-determined names and addresses.

SYMBOLIC CODING - In digital computer programming, any coding system using symbolic rather than actual computer addresses.

SYNCHRONOUS - Operation of a switching network by a clock pulse generator. All circuits in the network switch simultaneously, and all actions take place synchronously with the clock.

SYNTAX ERROR - An occurrence in the source program of a label expression, or condition that does not meet the format requirements of the assembler program.

TABLE - A data structure used to contain sequences of instructions, addresses, or data constants.

TRAILING EDGE - The transition of a pulse that occurs last, such as the high-to-low transition of a positive clock pulse.

TRANSITION - The instance of changing from one state to a second state.

THREE-STATE DEVICE or TRI-STATE DEVICE - A semiconductor logic device in which there are three possible output states: (1) a "logic 0" state, (2) a "logic 1" state, or (3) a state in which the output is, in effect, disconnected from the rest of the circuit and has no influence upon it.

THREE-BYTE INSTRUCTION - An instruction that consists of twenty-four contiguous bits occupying three successive memory locations.

TRUTH TABLE - A tabulation that shows the relation of all output logic levels of a digital circuit to all possible combinations of input logic levels in such a way as to characterize the circuit functions completely.

TWO-BYTE INSTRUCTION - An instruction that consists of sixteen contiguous bits occupying two successive memory locations.

TWO-PHASE CLOCK - A two-output timing device that provides two continuous series of timing pulse from the second series always following a single clock pulse from the first series. Depending on the type of two-phase clock, the pulses in the first and second series may or may not overlap each other. Usually identified as Phase 1 & Phase 2.

UNCONDITIONAL - Not subject to conditions external to the specific computer instruction.

UNCONDITIONAL CALL - A call instruction that is unconditional.

UNCONDITIONAL JUMP - A computer instruction that interrupts the normal process of obtaining the instructions in an ordered sequence and specifies the address from which the next instruction must be taken.

UNCONDITIONAL RETURN - A return instruction that is unconditional.

VLSI (VERY LARGE-SCALE INTEGRATION) - Monolithic digital integrated circuit chips with a typical complexity of two thousand or more gates or gate-equivalent circuits.

VOLATILE MEMORY - A semiconductor memory device in which the stored digital data is lost when the power is removed.

WEIGHTING - Most counters in the 7400 series of integrated circuit chips are weighted counters, that is, we can assign a weighted value to each of the flip-flop outputs in the counter. By summing the product of the logic state times the weighting value for each of the flip-flops, we can compute the counter state. For example, the weighting factors for a 4-bit binary counter are D = weight of 8, C = weight of 4; B = weight of 2, and A = weight of 1. The binary output, DCBA = 1101₂, from a 4-bit binary counter would therefore be 13.

WIRED-OR CIRCUIT - A circuit consisting of two or more semiconductor devices with open collector outputs in which the outputs are wired together. The output from the circuit is at a logic 0 if device A or device B or device C or is at a logic 0 state.

WORD - The maximum number of binary digits that can be stored in a single addressable memory location of a given computer system.

WRITE - In semiconductors and other types of memory devices - to transmit data into a memory device from some other digital electronic device. To WRITE is to STORE.

ZERO-PAGE - The lowest 256 address locations in memory. Where the highest 8-bits of address are always 0's and the lower 8-bits identify any location from 0 to 255. Therefore, only a single byte is needed to address a location in zero-page.

ZERO-PAGE ADDRESSING - The second byte of the instruction contains a zero-page address.

ZERO-PAGE INDEXED ADDRESSING - The second byte of the instruction is added to the index register (X or Y) to form a zero-page effective address. The carry (if any) is dropped.